This book was bound by

Badminton Press
18 Half Croft, Syston, Leicester, LE7 8LD
Telephone Leicester (0533) 602918

A Study Of Algorithms

For

Parallel Computers And VLSI

Systolic Processor Arrays

Volume – I

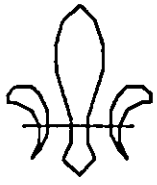
BY

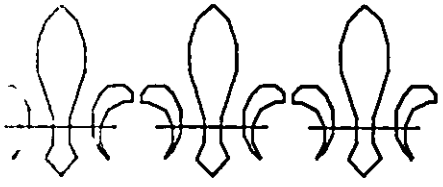
Michael P. Bekakos
B.Sc.(Hons.),M.Sc.,M.H.M.S.

A Doctoral Thesis
Submitted in partial fulfilment of the requirements
for the Award of Doctor of Philosophy
of the Loughborough University of Technology
July, 1986.

Supervisor: Professor David J. Evans
B.Sc.,M.Sc.,Ph.D.,D.Sc.,F.I.M.A.,F.B.C.S.
Department of Computer Studies

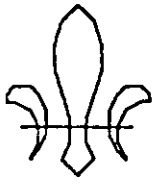
This Research is Sponsored by NATO and the Greek Ministry of National Economy under the Science Fellowship Contract No. [ΔΣ5450/TB.2391/15-7-81/ΑΥΣ-(ΑΕ.347/32213/Τροπ.6)].





Το Αφιερωνω

στον Πατερα μου



ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to Professor D. J. Evans, my supervisor, for his guidance, unfailing enthusiasm and patience to read an innumerable number of Thesis drafts.

I wish to thank N.A.T.O. and the Greek Ministry of National Economy, my scholarship bodies, whose financial support made the carrying out of this lengthy research possible.

To my main, morally and materially, supporters throughout all my postgraduate years, my parents, I wish to praise more than can be expressed, for their understanding and all the sacrifices they went unquestionably into in order to see me achieving my goals.

I wish to especially acknowledge Mr. Robert P. Stallard for all his support and constructive criticisms on the NEPTUNE Parallel Computer Prototype.

Last, but by no means least, I am grateful to my wife Christina, who patiently accepted the long hours of absence during the 'ups and downs' of this work and took care of marriage and family so that I could pursue my degree.

Finally, my son Panayiotis kept smiling at me whenever I needed it the most.

I am also indebted to Miss J. M. Briers whose artistic professionalism in typing is reflected from the script itself.

TITLE:

A STUDY OF ALGORITHMS FOR PARALLEL COMPUTERS AND VLSI SYSTOLIC PROCESSOR ARRAYS

ABSTRACT

In this Thesis the design and analysis of parallel algorithms is investigated under the framework of, either, being suitable for execution on asynchronous Multiprocessor testbeds (*MIMD* organizations), or, due to the recent remarkable advance of 'Very Large Scale Integrated' - *VLSI* circuitry, of being suitable for direct hardware implementation.

In the first three introductory *Chapters* a brief and taxonomically disciplined *state-of-the-art* survey is presented with up-to-date information on the parallel computing environment. This survey is relatively complemented by the contents of the last *Chapter VIII*, where most of the envisaged technological advancements are discussed.

More analytically, *Chapter I* is devoted to the overview of parallel computer systems and prototypes. After the exploitation of parallelism in various parallel computer structures, in terms of classifying the various architectural designs, the *Chapter* continues with the genealogical taxonomy of the main current multiple processor complexes.

In *Chapter II* the programming tools and algorithms to exploit the parallel hardware potential are introduced. In particular, concurrent programming languages motivations and general concepts for parallel processing are discussed, to continue with various methodological design and analysis aspects of parallel algorithms to appropriately map onto the different architectural categories.

In both these *Chapters* particular reference has been made to the *'NEPTUNE' MIMD* prototype, sited at the Department of Computer Studies, at Loughborough University of Technology, on which the bulk of the experimental work contained herein was carried out.

Developments in microelectronics have revolutionized computer design. *VLSI* technology has enormously increased the number and complexity of components that can fit on a chip. As a result, machines-on-a-chip have emerged; these machines can be used as special-purpose devices attached to a conventional 'host' computer. In *Chapter III*, at first, various computational models and 'Knowledge Information Processing Systems' - *KIPS* are introduced, to continue with the embedding of information flow schemes on grids and in *VLSI* chip area and time.

An extensive investigation on the potential parallelism of a new powerful class of Group Explicit methods compared to the Standard Explicit method is carried out in *Chapter IV*, for the solution of parabolic partial differential equations. For the performance analysis, on the provided MIMD testbed, of all parallel implementations in the Thesis, a detailed 'Deterministic Performance Model' - *DPM* is established along with all the particular general formulae for the estimation of its various parameters.

A complete performance exploitation of the *'NEPTUNE' MIMD*

prototype is pursued in *Chapter V*, by implementing several parallel algorithms using the Cyclic Odd-Even reduction technique, in combination with all the possible parallel constructs for the system, to solve Toeplitz tridiagonal linear systems for use in signal and image processing applications. The *Chapter* continues with the implementation of several new parallel algorithms using the same technique, to solve general periodic and non-periodic tridiagonal linear systems, following an alternate approach for the utilization of any number of processors.

In *Chapter VI* the research is being concentrated on algorithmically specialized *systolic* networks. A new powerful 'rotating' and 'folding' technique is introduced and applied on two-dimensional systolic communication geometries to solve a variety of occurring problems. In particular, at first, the matrix-vector and matrix multiplication problems are treated; then, the method is applied to tridiagonal and quindiagonal linear systems, and eventually generalized for p semi-bandwidth linear systems. To bypass the complexity arising along with the increase of the semi-bandwidth of the coefficient matrix, an alternative 'unidirectional' factorization of the central formatted submatrix is proposed and exemplified.
The resulting upper and lower triangular linear systems are solved again by the new method using a linear systolic array of processors.

Finally, in *Chapter VII*, single stage computational *dewavefronts* are investigated, as an expansion of the 'rotate' and 'fold' method, for the implementation of the 'Quadrant Interlocking Factorization' - *QIF* parallel method on a data-driven 'Wavefront Array Processor' - *WAP*, for the case that the coefficient matrix of the linear system is a

compact dense (nxn) matrix.

The Thesis is concluded with general comments on future computer architectures, overviewing conclusions and a discussion for further future research topics in this area. *References* and *Appendices* with complementary theory and proofs, where needed, and a selection of optimized parallel computer programs from our experimental work are also included. ■

CONTENTS

TABLE OF CONTENTS

A Tree-like Logical Subdivision Of Chapter III Contents

A Tree-like Logical Subdivision Of Chapter IV Contents

VOLUME - II

A TREE-LIKE LOGICAL SUBDIVISION OF CHAPTER V CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

Volume I

CHAPTER I: AN OVERVIEW OF PARALLEL COMPUTER ARCHITECTURES

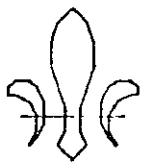
- *SECTION A:* EXPLOITATION OF PARALLELISM IN VARIOUS PARALLEL COMPUTER ARCHITECTURES
  - *I.A.1:* THE INNOVATION OF THE PARALLEL NOTION
  - *I.A.2:* CLASSIFICATION OF DESIGNS
    - *I.A.2.1:* FLYNN'S VERY HIGH-SPEED COMPUTING SYSTEMS
    - *I.A.2.2:* SHORE'S TAXONOMY
    - *I.A.2.3:* OTHER CLASSIFICATION APPROACHES
- *SECTION B:* THE MAIN CURRENT MULTIPLE PROCESSOR ARCHITECTURES
  - *I.B.1:* INTRODUCTION
  - *I.B.2:* THE GENEALOGY OF THE SIMD ORGANIZATION
  - *I.B.3:* THE UTILIZATION AND APPLICATION OF THE SIMD SYSTEMS
    - *I.B.3.1:* THE ASSOCIATIVE PROCESSOR ARCHITECTURE
      - *I.B.3.1.1:* THE ASSOCIATIVE MEMORY ORGANIZATION
      - *I.B.3.1.2:* ARCHITECTURAL TAXONOMY OF THE ASSOCIATIVE PROCESSORS
        - *I.B.3.1.2.i:* FULLY PARALLEL ASSOCIATIVE PROCESSORS
        - *I.B.3.1.2.ii:* BIT-SERIAL ASSOCIATIVE PROCESSORS
        - *I.B.3.1.2.iii:* WORD-SERIAL ASSOCIATIVE PROCESSORS
        - *I.B.3.1.2.iv:* BLOCK-ORIENTED ASSOCIATIVE PROCESSORS
        - *I.B.3.1.2.v:* HIGHLY PARALLEL ASSOCIATIVE PROCESSORS
    - *I.B.3.2:* PARALLEL PROCESSOR ARCHITECTURES
      - *I.B.3.2.1:* THE UTILIZATION OF PARALLEL MEMORIES
      - *I.B.3.2.2:* THE INTERCONNECTION NETWORKS
      - *I.B.3.2.3:* IMPLEMENTED PARALLEL PROCESSOR SYSTEMS
        - *I.B.3.2.3.i:* THE ORTHOGONAL COMPUTER CONCEPT
  - *I.B.4:* THE GENEALOGY OF THE PIPELINED VECTOR ORGANIZATION
    - *I.B.4.1:* PIPELINE AS A FUNDAMENTAL DESIGN PRINCIPLE AND PERFORMANCE CHARACTERISTICS
    - *I.B.4.2:* VECTOR PROCESSING CHARACTERISTICS
    - *I.B.4.3:* IMPLEMENTED PIPELINED VECTOR COMPUTERS
  - *I.B.5:* THE MIMD MULTIPROCESSING ARCHITECTURES
    - *I.B.5.1:* THE MIMD HARDWARE SYSTEM ORGANIZATION
      - *I.B.5.1.1:* THE TIME-SHARED/COMMON BUS INTERCONNECTION SCHEMA
      - *I.B.5.1.2:* THE CROSSBAR SWITCH MATRIX INTERCONNECTION SCHEMA
      - *I.B.5.1.3:* THE MULTIBUS/MULTIPORT INTERCONNECTION SCHEMA
      - *I.B.5.1.4:* THE VIRTUAL AND MAILBOX LOGICAL INTERCONNECTION SCHEMAS
    - *I.B.5.2:* THE MIMD OPERATING SYSTEM ORGANIZATION
    - *I.B.5.3:* IMPLEMENTED MIMD ARCHITECTURES
      - *I.B.5.3.1:* THE INTERDATA DUAL PROCESSOR SYSTEM
      - *I.B.5.3.2:* THE '*NEPTUNE*' SYSTEM
  - *I.B.6:* A GENERAL REVIEW OF MULTIPLE PROCESSOR SYSTEMS' PRINCIPLE MOTIVATIONS

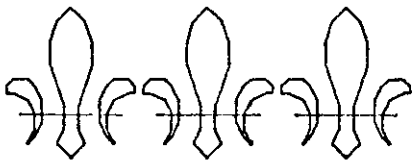
Chapter I

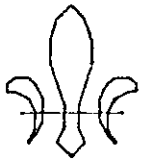
An Overview Of Parallel Computer Architectures

SECTION A

EXPLOITATION OF PARALLELISM IN VARIOUS PARALLEL COMPUTER ARCHITECTURES







'...the natural world is made up of millions of intelligent creatures interacting together in order to satisfy their requirements for life;

why then constrain a naturally parallel world into a series of actions which must strictly follow each other sequentially in time?'

From Professor D.J.Evans' lectures

I.A.1: THE INNOVATION OF THE PARALLEL NOTION

In this presentation of the process of evolution of the *parallel way of thinking*, we have not attempted a complete survey of the *conceived-implemented* parallel notions; we are roughly tracing the history of this development, laying out the principles that can be applied to classify these parallel architectures and characterize their relative performance.

As a consequence of the aim to efficiently use parallel systems, the programmer has to be aware of the overall structure of such a system and therefore, *Section B* is devoted to the description of the main classes of parallel architectures.

A simple arithmetic operation, i.e. a floating-point multiplication, presented schematically in *Figure (I.A.1-f1)*, demonstrates a ten-fold arithmetic computing speed increment, every five years, since the first commercially established computer, the *UNIVAC 1*, in 1951 (see Hockney [*HOCK81*], in 1981). The combination of all the technological improvements in the performance of the hardware components, with the introduction of even greater parallelism at all levels of the computer architecture, has made possible this significant increment in computational speed.

The earliest reference to parallelism, in computer design, is thought to be in General L.F. Menabrea's[†] publication, in the Bibliothèque

[†] *An Italian army officer, who sat through a series of lectures Babbage gave in Turin in 1840.*

Universelle de Genève, October 1842, entitled 'Sketch of the Analytical Engine', invented by Charles Babbage. There, listing the utility of the

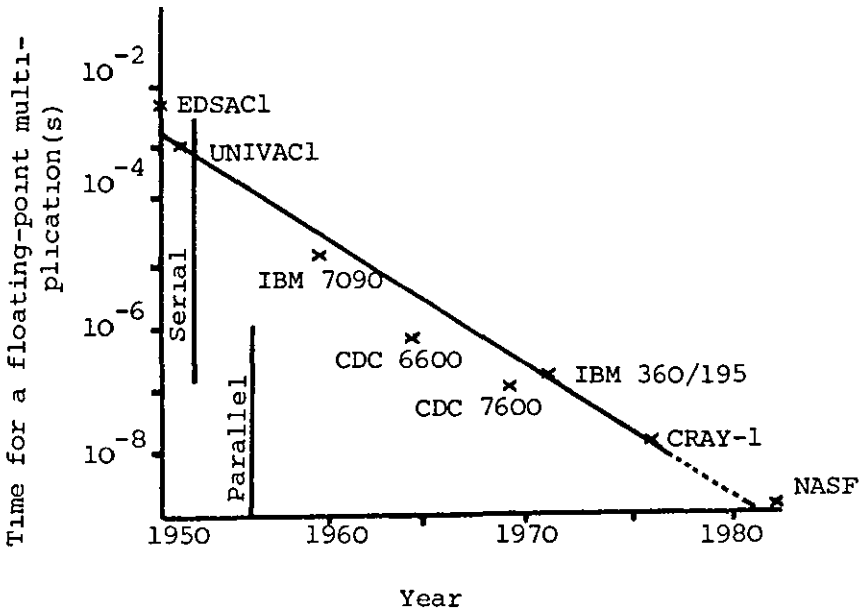


Figure I.A.1-f1: The Increment in Computational Arithmetic Speed (a factor of 10 every 5 years).

analytic engine, he writes: 'Secondly, the economy of time: to convince ourselves of this, we need only recollect that the multiplication of two numbers, consisting each of twenty figures, requires at the very utmost three minutes. Likewise, when a long series of identical computations is to be performed, such as, those required for the formation of numerical tables, 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 processes'.

Although Babbage's notion was neither implemented in the final design of his calculating engine or elsewhere, due to the lack of technological development accordingly, though, the notion of the *parallel way of thinking* had been conceived. A search in the evolution of the electronic technology, on the hardware of the computer systems, reveals

the *reason* which finally urged scientists to this notion, as the only, up-to-date, way to overcome the existing computing difficulties and constraints.

The *first-generation* of computers, in 1950s, according to the technological achievements, used vacuum tubes, magnetic drums as central memories and electronic valves as their switching components, with *gate delay* times of approximately 1μs.

The use of the discrete germanium transistors, around 1960, with *gate delay* times of approximately 0.3μs, gave rise to the *second-generation* computers, such as the *IBM 7090*.

The *third-generation* of computers, which were introduced around 1965, used bipolar planar Integrated Circuits *(ICS)* on Silicon, at a Small-Scale Integration *(SSI)* level, with a few gates per chip and *gate delay* times of about 10ns, and later, around 1975, of slightly less than 1ns. The increase in the processing rate of a computer, was the aim of all these developments in device, circuit technology and miniaturization techniques, through these computer generations.

The processing rate of a computer, primarily, can be conceived as the volume of the cube(s) enclosed by one or more points in a rectangular coordinate system, with three axes denoted by the *clock rate*, the *word width*, and the *number of words*. Consequently, the processing rate of a computer could be improved by increasing either the *clock rates* in the system or the number of bits that could be processed simultaneously.

The use of high-speed switching devices and circuits, as well as the shortening of the propagation time in interconnection lines, could increase the *clock rates*. However, as improvements in switching devices and miniaturization reached the current engineering and electronic limits,

and as word width was limited by the precision of intended applications, it was apparent that any further significant increase in computing speed would be obtained by the *concurrent* processing of a number of words.

A decisive factor in this turn to *concurrent* processing was the development of the semiconductor technology, which used the alternative Metal Oxide Silicon *(MOS)* and although five to ten times slower, offered much greater packing densities. Thus, around the beginning of 1980s, the construction of microprocessors started, with speeds and capacities similar to *first generation* computers, but on a few millimetres square size of a single silicon chip. This fact introduced the *fourth generation* of computers, remained in theory up to then, by implementing various highly parallel architectures, combining different numbers of processors, as well as, new scientific terms came to use, such as multiprogramming, multiprocessing, on-line, real-time, synchronous, asynchronous, etc.

All these various multiple processor architectures can be categorized in four distinct(!) organizations - *Associative*, *Parallel*, *Pipelined*, and *Multiprocessors* – of which the architectural structure and processing techniques will be surveyed.

An attempt to summarize the principal ways to introduce the notion of *parallel processing* at the hardware level of the various computer architectures, results in:

1. *Pipelining* - assembly-line techniques are employed to improve the performance of the arithmetic or the control unit, by decomposing a repeated sequential process into subprocesses, capable of being executed by dedicated autonomous units;
2. *Functional* - various independent units are provided, to perform different functions, i.e. logic, addition or multiplication, allowing simultaneous operation on different data;

3. *Array* - an array of identical Processing Elements (PEs) is provided, capable of performing, simultaneously, the same operation on different sets of data, stored in their private memories, under common control - i.e. lockstep operation; and,
4. *Multiprocessing* - several processors communicate via a common or *shared* memory, each obeying its own instructions.

Naturally, we may combine some or all of these parallel features on individual designs; for example, a Processor Array may have pipelined arithmetic units as its processing elements, and a functional unit, in a multi-unit system, could be a Processor Array.

It would be impractical to present a comprehensive description of all the designs in the above categories; instead, we have selected the principally significant architectures, which differ sufficiently from each other, to illustrate alternative hardware and software approaches. Specifically for the Multiprocessor class, the *NEPTUNE* parallel processing system, at Loughborough University of Technology, is described in greater detail, due to the fact that it was extensively used during the carrying out of the present research.

Finally, and as a preface for *Chapter III*, we ought to mention here that due to the previously mentioned lack of synchronization between the theoretical research achievements and the technology, although at the first steps of the *parallel fourth generation*, with a tremendous number of physical problems still not yet implemented, we are talking about the 'Fifth Generation of Computer Systems'- *FGCS*, which will represent the unification of research into *VLSI* processors and into distributed processing.

In this generation each computer system will consist of a network

of computing elements supporting an individual application or need. *VLSI* processor research will allow a computing element to provide, either a general-purpose or a special-purpose function, and range in power from a main-frame computer to a miniature microcomputer.

On the other hand, distributed processing research will allow a network to be physically dispersed across a country or a building, or to be physically close, as on a single highly integrated chip.

More about this generation of computers, as well as, about the general *data flow* computer concepts, i.e. a computer which allows the execution of programs represented in data-flow form, to achieve highly parallel computation, will be discussed in *Chapter III*.

In conclusion, we would like to suggest that this work, somehow, is related to both previous generations, by presenting algorithms implemented on parallel architectures of *fourth generation* and proposing the *direct* hardware implementation of others using Systolic *VLSI* processor arrays.

I.A.2 Classification Of Designs

Any attempt to strictly classify all the proposed computer architectures, or at least those which have been already well established, would not be wholly successful, mainly due to the fact that some of them (e.g. *ICL DAP*), could fit equally well into several different classified groups, or others (e.g. Pipelined computers) would not fit in any of them.

Alternatively, on the one hand, we shall briefly present the theoretical concepts of the architectures taxonomy given by different researchers, especially by the two pioneers, Flynn [*FLYN66*] in 1966 and Shore [*SHOR73*] in 1973, since both proposals, not only, have been widely mentioned but also the corresponding terminology has contributed to the formation of the computer science language; whilst, on the other hand, in *Section B* we shall refer, in greater detail, to the four specific organizations previously mentioned, i.e. *Associative*, *Parallel*, *Pipelined* and *Multiprocessors*, presenting well established corresponding architectures with their processing techniques and features.

I.A.2.1: Flynn's Very High-Speed Computing Systems

When Flynn presented his theoretical concepts for the classification of parallel organizations, these were not based on the hardware structure of the computers, but on the dependent relation between the instructions being propagated by the computer and the data being processed.

For convenience he adopted two new definitions: the *Instruction Stream*, defined as the sequence of instructions which are to be performed by the system, and the *Data Stream*, as the sequence of data called for, by the instruction stream (including any input and partial or temporary results). Also, he defined two additional useful notions: the *Bandwidth* notion, by

which he expressed the time-rate of occurrence, and the *Latency* or *Latent period* notion, as the time needed, totally, from excitation to response, in order to process a particular data unit, during a phase in the computing process.

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

By using the first two definitions, Flynn categorized the existing (mostly theoretically) organizations depending on the *multiplicity* of the hardware provided, to service the Instruction and Data Streams, thus avoiding the ubiquitous and ambiguous term *parallelism*.

By the word *multiplicity*, Flynn meant the *activity* of the *most constrained* component of the architecture, i.e. the maximum possible number of *simultaneous* operations (instructions) or operands (data), which that constrained component was capable of executing in the same execution phase.

Consequently, four broad(!) classifications emerged, being characterized from the multiplicity or not of the Instruction or Data streams:

1. Single Instruction stream - Single Data stream *(SISD)*;
2. Single Instruction stream - Multiple Data stream *(SIMD)*;
3. Multiple Instruction stream - Single Data stream *(MISD)*;
4. Multiple Instruction stream - Multiple Data stream *(MIMD)*;

The first class of the confluent, (as *confluence* or *concurrence* he named the ratio of the number of simultaneous instructions being processed by the constrained multiplicity, i.e. the tightest constraint imposed by a system component), *SISD* processor [like *IBM STRETCH; CDC 6600*

(unpipelined); *CDC 7600* (pipelined arithmetic); *IBM 360/90* series] is nothing more than a conventional serial von Neumann computer, which achieves its power by overlapping the various sequential decisions involved in the execution of a single instruction. In *Figures (I.A.2.1-f1,f2)* we see a *SISD* organization, and the concurrency and instruction processing, respectively.

An increase in computational bandwidth is achieved by maximizing the utility of the constraining components (*bottleneck*), whereas, as it can be seen in *Figure (I.A.2.1-f2)*,a *chainly* sequential procedure is adopted to increase the computational speed. However, despite the various pipelining schemes invented to increase the efficiency of this organization, the bottleneck or constraint of decoding only one instruction in a unit time has remained.

The *SIMD*-type structures, proposed by Unger [*UNGE58*] in 1958, Slotnick, et al [*SLOT62*] in 1962, Crane and Githens [*CRAN65*] in 1965, and Hellerman [*HELL66*] in 1966, possess a single stream of instructions and n (generally) execution vector elements. Each vector element is considered as belonging to a discrete data stream, excluding degenerate extreme cases (e.g. vectors of length one), thus there are multiple data streams, on which the single instruction stream acts simultaneously, without any concurrent techniques. Although the latency in the data stream, due to data communication problems, causes some difficulties, the performance of this organization can be increased by using longer vectors of execution units.

The *MISD*-type structures, the outline of which can be seen in *Figure (I.A.2.1-f3)*, is apparently the least favourable class compared to the others, since no examples of any established organizations exist, the

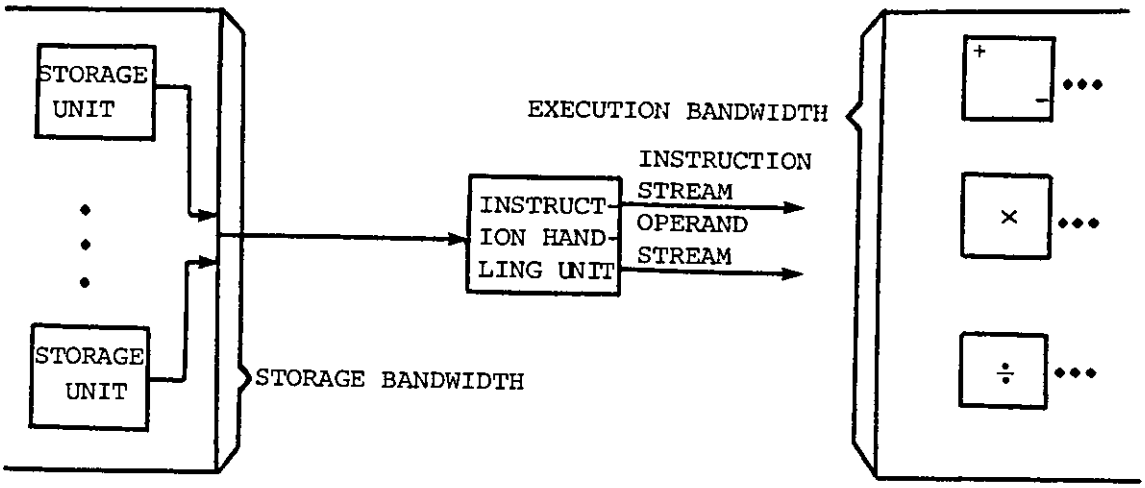


Figure I.A.2.1-f1: Flynn's *SISD* Organization.

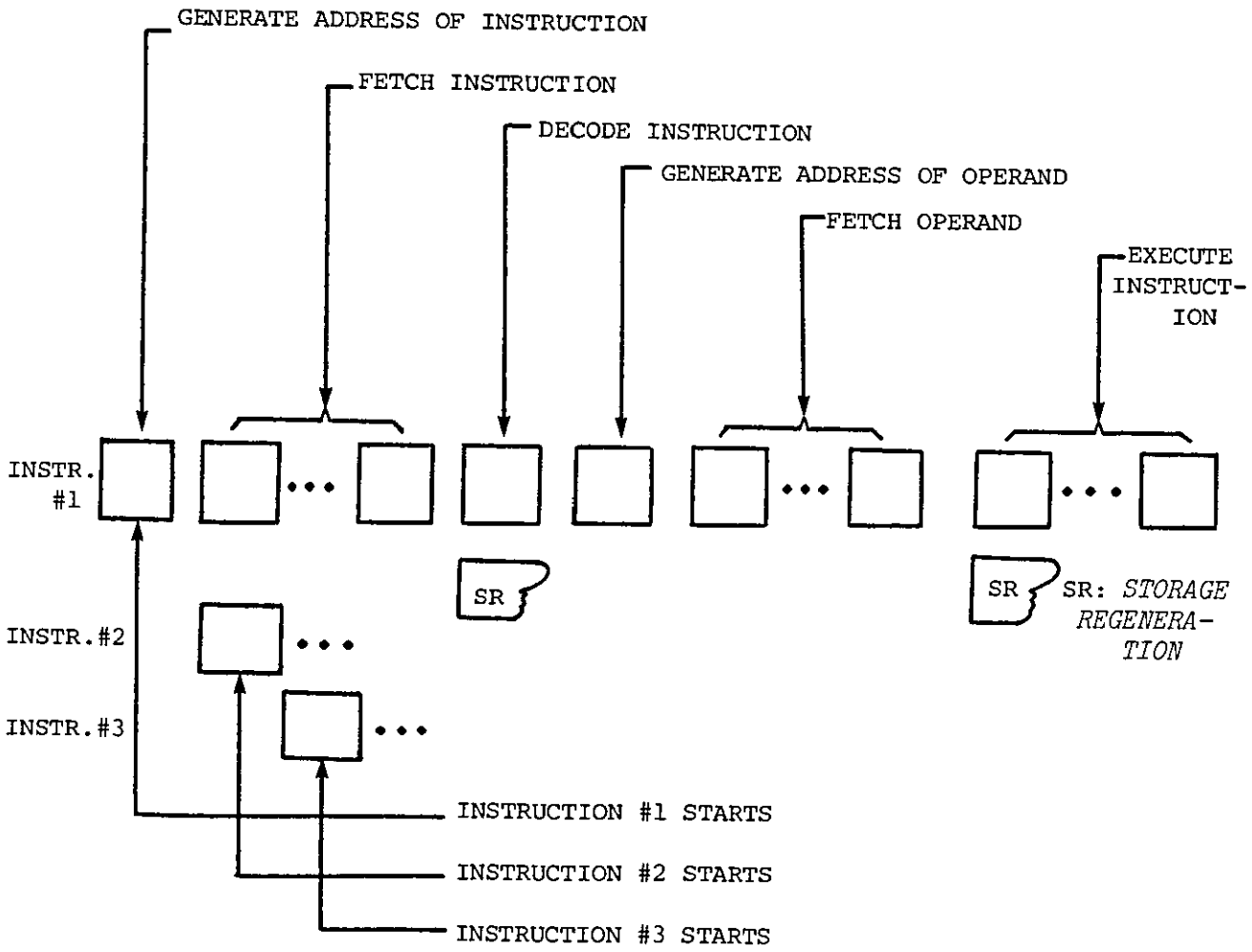


Figure I.A.2.1-f2: Concurrency and Instruction Processing.

nearest example being the *line printer*. This class has employed a forwarding procedure of the data flowing through the execution components. The instruction that any control component executes may be *fixed* (thus the interconnection of the components must be flexible), *semifixed* (such that the function of any component is fixed for one pass of a data file), or *variable* (meaning that the execution of a stream of instructions may take place at any point along the single data stream). According to this order of the execution components, only the first of them faces the source data stream, whereas the rest operate on the resultant data stream from the previous, in sequence, component.

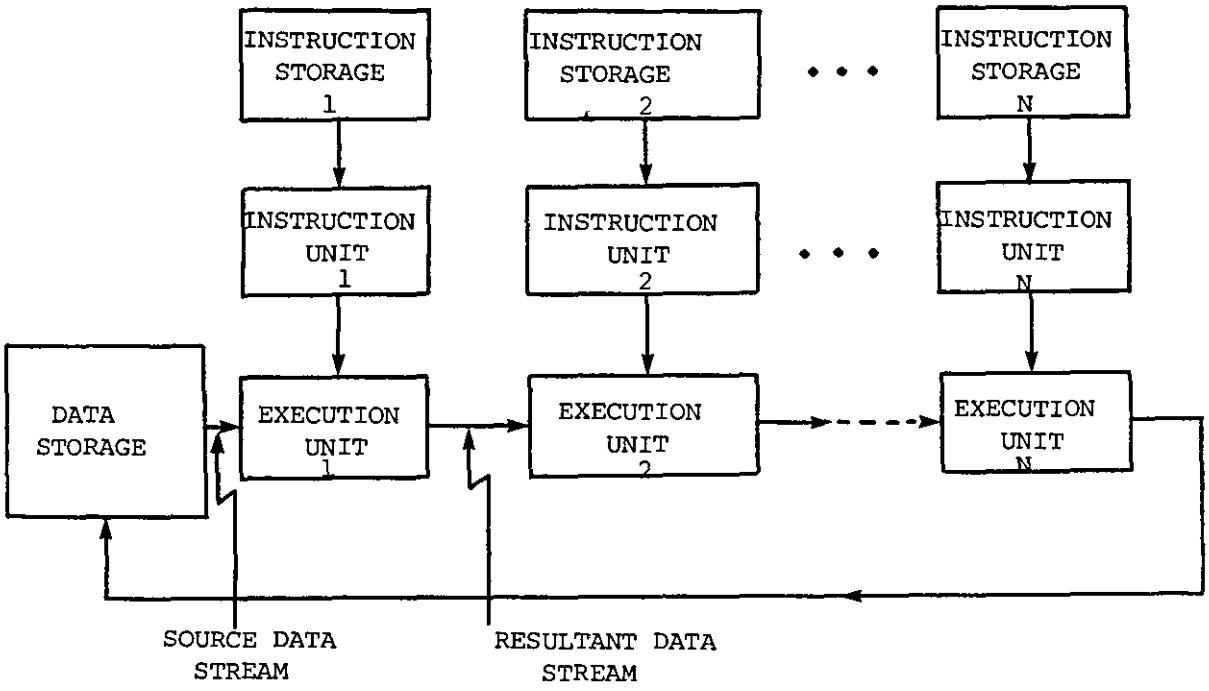


Figure I.A.2.1-f3: A *MISD* Organization.

In the last class of the *MIMD* organizations, we have several instruction processing components and corresponding data streams. These components (*processors*) are independent of each other and capable of executing instructions simultaneously. They each have arithmetic and

logic capability, but communication is limited by their need to *build a path*, to reach the appropriate required data. This class includes all the different Multiprocessor organizations†, starting from the linked main-frame systems, up to large arrays of microprocessors.

Recapitulating, Flynn's taxonomy is somehow obscure, due to the fact that it was dependent on the systems interrelations, between instructions and the data to be processed, rather than the architectural design of the parallel computers; consequently there is no significant distinctive point between these classes (the *MIMD* class exempted), e.g. the Pipelined and the Processor Array computers are considered similar, although they are entirely different architectures.

More or less, according to Flynn, all parallel computers, except the Multiprocessors, could be lumped into the *SIMD* classification. Also, the meaning of the *Data Streams*, as used by Flynn, caused many ambiguities, which led, in some cases, to confusion; for example, although Flynn had classified the Pipelined vector computers into the *SIMD* class, the confusion led scientists, either to place them in the *SISD* class (see Hockney [*HOCK77*] in 1977), because they were processing a single stream of vectorized data, or others (see Gorsline [*GORS80*] in 1980) accepted Flynn's classification, but also considered that these systems could fall, evenly well, in the Multiprocessor class; this was due to the fact that they had a pipelined arithmetic unit which could be considered equivalent to performing simultaneously various instructions, on either, a single vectorized data stream or a multiple scalar stream.

Consequently, in *Section B* of this *Chapter*, we shall consider the *SIMD* and *Pipelined* computers as two distinct classes according to their architecture, along with the *Multiprocessor* category.

---

† *The NEPTUNE parallel computer system at Loughborough University is an example of this class.*

I.A.2.2: SHORE'S TAXONOMY

Shore [*SHOR73*] in 1973 attempted a classification of the parallel architectures, based on their constituent hardware components. According to him, all existing computers could belong to one of the six different types of machines he proposed, which can be seen in *Figure (I.A.2.2-f1)*.

The Machine *I*, is the conventional serial von-Neumann organization, consisting of an Instruction Memory (*IM*), a single Control Unit (*CU*), a Processing Unit (*PU*) and a Data Memory (*DM*). The processing unit, may consist of multiple functional components, pipelined or not, and follows the serial way of word reading, i.e. all bits of a single word are read in order to be processed simultaneously. Examples of this type are the *CDC 7600*, a Pipelined Scalar computer, as well as the *CRAY-1* , a Pipelined Vector computer.

A change on reading the data, from all bits of a word, to a bit from all words in the memory, i.e. bit-serially, but word processing in parallel, yields Machine *II*, which in all other ways is a replica of Machine *I*. More schematically, 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 vertical slices*.

A combination of Machines *I* and *II*, yields Machine *III*; this means that Machine *III* has two processing units, one horizontal and one vertical, so the processing can be done in either of the two directions. If the *ICL DAP* or *STARAN* computers had separate processing units to offer this capability, they could be placed in this class of computers. An example of this organization is the Sanders Associates *OMEN-60* series of computers (see Higbie [*HIGB72*], in 1972).

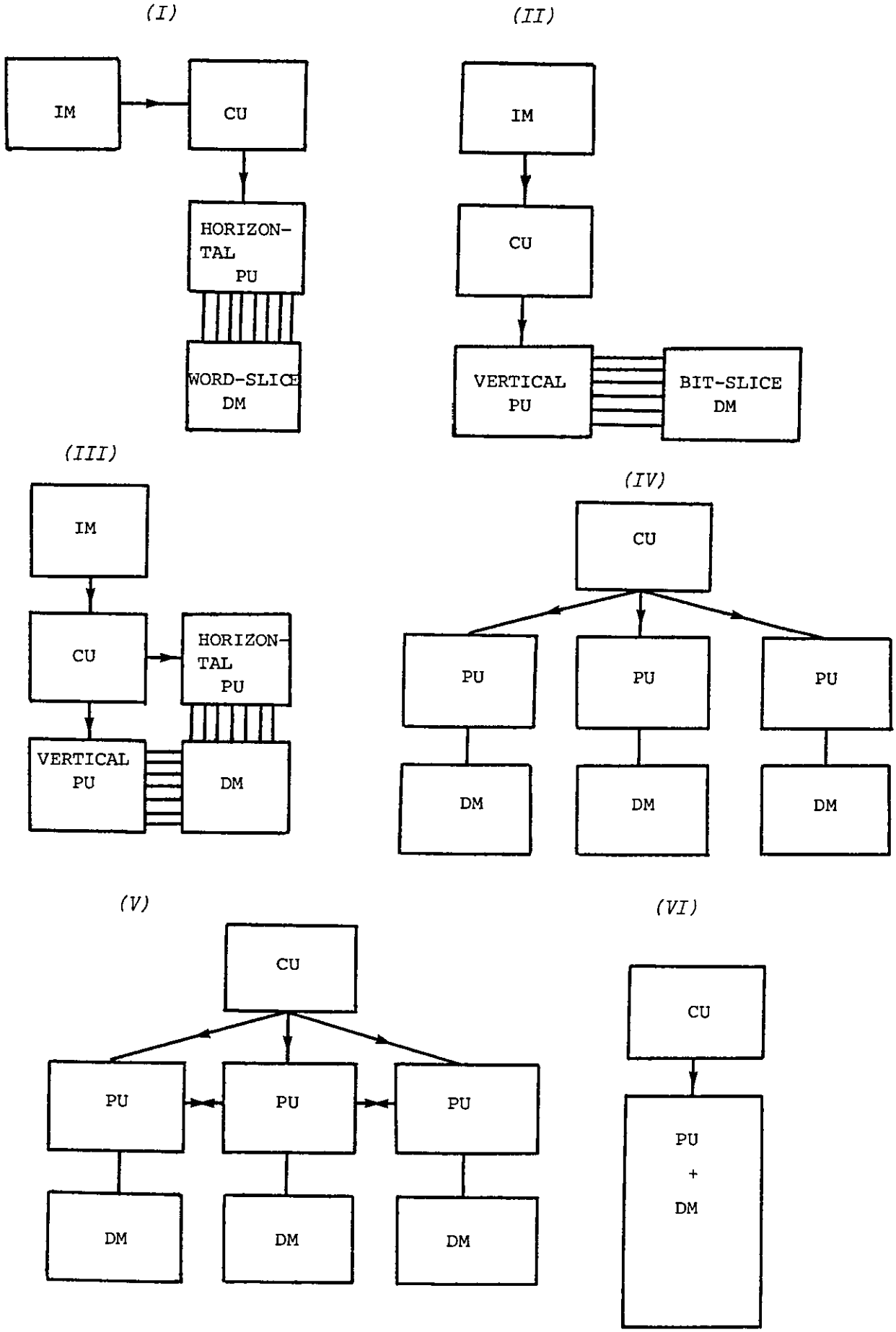


Figure I.A.2.2-f1: The Configuration of the Six Machine Classes.

The Machine *IV*, consists of a single control unit and as many as possible, independent to each other, processing elements. Each of these elements consists of a processing unit and a data memory. The communication between these elements takes place only through the control unit. A well known example of this Machine is the *PEPE* system.

The additional facility to this Machine of communication between the nearest-neighbour - in a line - processing elements, and thus the capability of each element to access data not only from its own private memory but also of its immediate neighbours, gave rise to Machine *V*. An example of that Machine is *ILLIAC IV*, which, in addition, provides a short-cut communication every eight processing elements.

Shore's last class of organization is Machine *VI*, or otherwise Logic-In-Memory Array (*LIMA*). The difference between this Machine and the previous ones is that the processing units and the data memory are no longer individual hardware parts, communicating through data buses or switching devices, but they are on the same *IC* board. Examples range from simple associative memories to complex associative processors.

Generally speaking, comparing this classification with Flynn's one, we observe that, on the one hand, less significantly, the numerical designator for the characterization of each class is not very mnemonically successful and,on the other hand, Shore's classification does not offer anything new, but only a subcategorization of the obscure *SIMD* class given by Flynn, except for Machine *I* which is of *SISD* type.

Finally, again the Pipelined computers are not distinguishably placed in a class representing their characteristics, but are mixed up with unpipelined Scalar computers.

I.A.2.3: Other Classification Approaches

In this paragraph we refer to some other classification approaches, less significant than the former two, based mainly upon the notion of parallelism.

One of these taxonomies was suggested by Hobbs, et al [*HOBB70*] in 1970 and distinguishes the parallel architectures into Multiprocessors, Associative processors, Network or Array processors and Functional machines.

Furthermore, Hobbs, et al proposed that architectures could be classified based upon the amount of parallelism involved in the Control, Data streams, and Processing units; but these factors proved to be vague since they are present in all highly parallel machines.

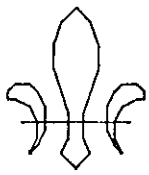
On the other hand, Murtha and Beadles [*MURT64*] based their taxonomy view upon the parallelism properties, attempting to underline the differences between the Multiprocessors and the Highly parallel organizations. According to them the parallel organizations could be classified into, the General-purpose network computers, the Special-purpose network computers with global parallelism and finally the Non-global, semi-independent network computers with local parallelism - thus including in this class all the computers excluded from the former two classes.

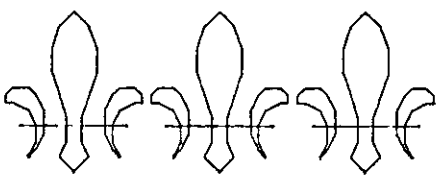
Furthermore, they distinguished some subclasses for the first two classes, the General-purpose network computers subclass, with a centralized common control, and the General-purpose network computers subclass, with identical processors but independent instruction execution actions; regarding the second class, they considered the Pattern processors and the Associative processors subclasses.

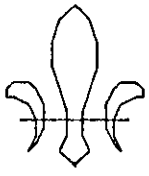
These two classifications of the computer systems together with the previous two, are the most significant and widely known attempts to categorize the different computer architectures into groups and consequently they will provide the framework within which to view the *Associative*, *Parallel*, *Pipelined* and *MIMD* organizations.

SECTION B

THE MAIN CURRENT
MULTIPLE PROCESSOR
ARCHITECTURES







'Anyone who says he knows how computers should be

built should have his head examined!

The man who says it is either inexperienced or

really mad.'

Computer Architecture

James E. Thornton

I.B.1 INTRODUCTION

It is certainly true that the terms *parallel computers* and *parallel processing* have been used in many ways in the previous section; starting 30 years ago, when it referred to computers performing arithmetic operations on whole words, rather than on one bit at a time, up to recent years, where we talked about Multiple-processors, Array-type machines (Parallel or Pipelined), Associative processors, etc.

In this section, we start from a basic idea, the classification of all the computer architectures into broad consistent categories, each of which has certain hardware and software criteria, which must be obeyed by an architecture in order to fall into this category. In consequence, and inside each of these categories, we shall distinguish subcategories, where it is applicable, according to the particular characteristics of the architectures included in the main category.

The basis to this classification will be the previously discussed (see par.- *I.A.2.1*) Flynn's classification in conjunction with Higbie's [*HIGB73*] expansion of Flynn's *SIMD* category.

Higbie distinguished four subcategories in the *SIMD* class:

1. *The Array Processor;*

2. *The Associative Memory Processor;*

3. *The Associative Array Processor;* and

4. *The Orthogonal Processor.*

The difference between the first two subcategories is that the data items are retrieved using the location addresses for the *Array Processors*, whereas for the *Associative Memory Processors*, depending on the particular characteristics of their memory, the data are addressed by a tag or their contents, rather than by their addresses. Also, the processing of data in the former processor is in parallel, whereas the latter does not require parallel operation, although the definition allows for machines that process data in this way.

The *Associative Array Processor* category, includes the processors which are associative, operating on arrays of data on a bit-slice basis, i.e. using a bit from every word in the memory.

The *Orthogonal Processor* consists of two subsystems, an array of processing elements to do the associative processing and a conventional processor system to do the sequential tasks.

According to Higbie, any computer organization comprising multiple arithmetic units, capable of operating on multiple data streams, is a parallel processor system; thus, he vaguely includes all of the above categories into this definition.

None of the classification schemes presented up to now was mathematically precise and a lot of confusion has been caused by using the current terminology.

Although we have mentioned (see par.- *I.A.1*) that we shall present a survey of four specific multiple processor organizations, *(Associative, Parallel, Pipelined* and *Multiprocessors)*, the first two organizations

consist of subclasses of the general category of the *SIMD* systems. Consequently, summarizing we can say that we have three major architectural categories: the *SIMD*, the *Pipelined* and the *Multiprocessor* architectures.

In the following paragraphs of this section, we shall examine these categories in greater detail, referring to the most significant, widely known and commercially implemented examples of each class and their hardware and software characteristics, bearing in mind that the intersection of the above categories is not null due to the overlap in classifying machine architectures.

I.B.2: THE GENEALOGY OF THE SIMD ORGANIZATION

The creation of the subclasses of the *SIMD* category, is based on the generic relationships given by Thurber and Wald [*THUR75*] in 1975, but major modifications and subsequent amendments have been made, changes which will be justified, as we shall proceed further, by examining each of these subclasses individually.

Thurber and Wald, based their definitions and subdivision on Flynn [*FLYN72*] and Thurber [*THUR76*]; according to them the major characteristic of the *SIMD* category, is a single global control unit, which drives multiple processing elements, all of which either execute or ignore the current instruction.

They distinguished three subclasses , the *Associative Processors*, the *Parallel Processors* and the *Ensembles*. The justification for this subdivision, was based on the particular characteristic of the Associative memory when retrieving the data stored in it, for the first subclass, on the fact that the processing elements had a complexity of similar order to small computers, with a high level of interconnectivity amongst them, for the second subclass, and finally, on the fact that in the *ensemble* architectures the level of interconnectivity could be non-existent or very low.

In *Figure (I.B.2-f1)* we can see the generic relationships which we present in this *Section* and which will be justified in the subsequent examination of each subclass. It should be understood that *SIMD* processors are special-purpose and useful for a limited set of applications, which will be summarized in the following paragraph; in addition, the reasons for which we need the *SIMD* processors will be surveyed.

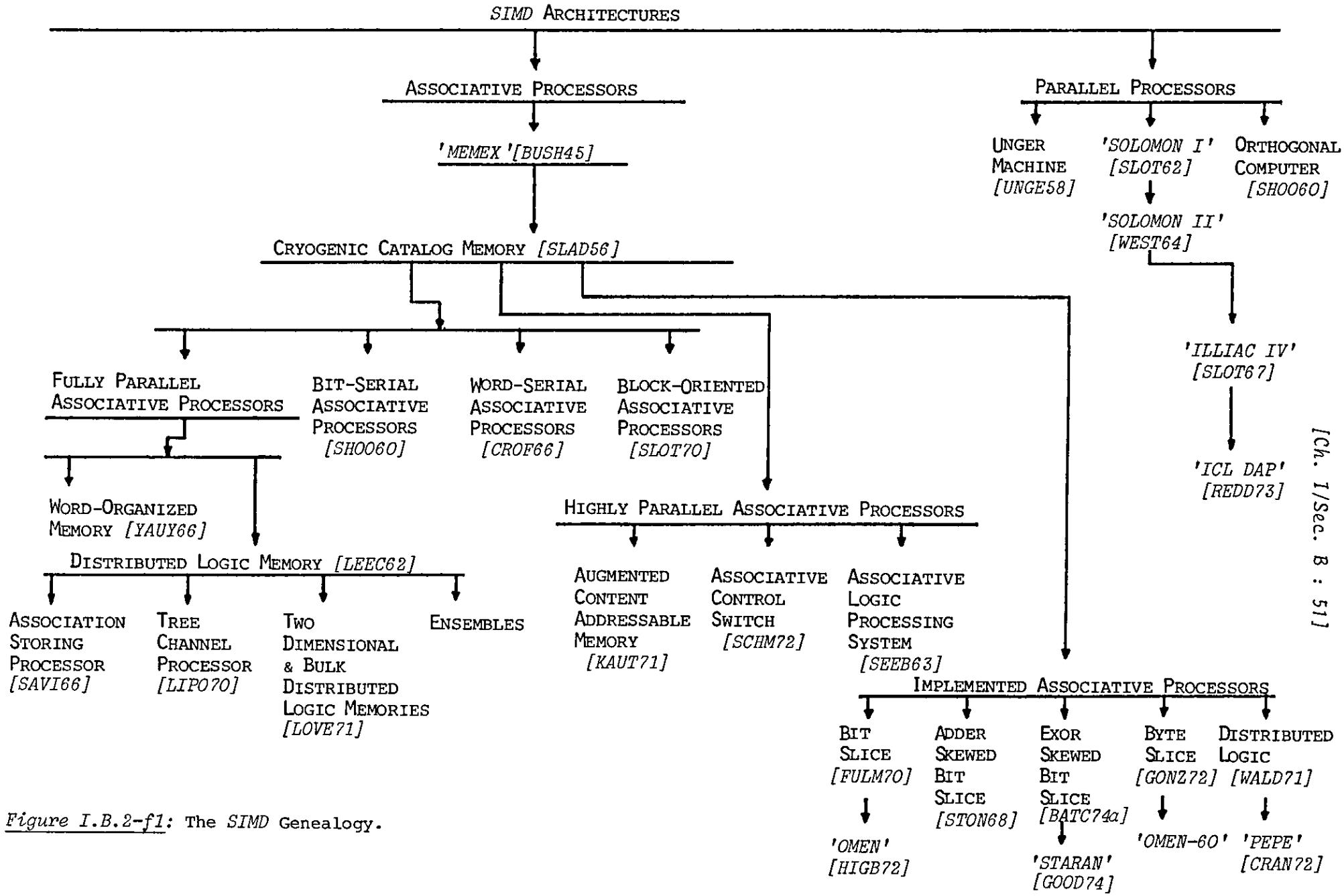


Figure I.B.2-f1: The *SIMD* Genealogy.

I.B.3: The Utilization And Application Of The SIMD Systems

The most important aspects for utilizing *SIMD* architectures can be categorized in the following three classes, although, we must bear in mind that, these systems are special-purpose computers and an improper application of them could lead to highly inefficient results.

The *first* class can be characterized from the *hardware* structure of these systems, which is more efficient, especially now with the advent of *LSI* microprocessors, compared to other Multiprocessors, for problems with large amounts of parallelism, providing the economy of duplicating structures and lower nonrecurring and recurring costs.

The characterization of the *second* class depends on the *software* used in these systems and which tends to be simpler, with less executive function requirements, than the one needed for Multiprocessors, thus making the construction of large systems easier.

Finally, in the *third* class comes the *functional* utility of these machines, which proves to be more efficient than the Multiprocessors for large problems demanding heavy data processing, e.g. weather forecasting, and for problems inherently possessing global parallelism, providing at the same time reliability, simpler system complexity and considerably higher potential computational load.

On the other hand, the categorization of the problems on which a cost effective implementation of the *SIMD* systems may be possible, has concentrated all the scientists' research attention. However, it is not always enough just to describe and analyze the nature of the problem and then offer a solution; it must proceed further, with an actual implementation, which of course depends on the economics of the problem. This means that researchers must not purely develop solutions for problems (which sometimes

may be *ill-defined*), but also must concentrate on the systems aspects of the problem too.

Numerous applications have been proposed for Associative and Parallel Processors, the most common being for matrix multiplication, differential equations and linear programming; some of these applications are quite well suited for the former class whilst others for the latter class.

By utilizing *SIMD* systems, the elimination of critical bottlenecks, which had appeared in the general-purpose computer systems, seemed to be apparent. However, the Associative Processors were constrained by high cost-factors, making feasible only the use of small associative memory systems for various, but limited size problems, such as, in the management of computer resources involving protection mechanisms, resource allocation, etc. Of course things are going to change with the evolution of *LSI* technology, but in the mean time, due to these cost-factors the burden of implementing this category of systems was carried out, mostly, by using Parallel Processors, which were applied in various research areas, such as, weather forecasting, nuclear data processing, ballistic missile defence, etc.

An attempt to summarize the numerous applications that the *SIMD* systems have been proposed to be well suited for, distinguishing between Associative and Parallel Processors, would give the following; the applications appearing in the *Parallel Processors' menu* to be very cost-effective are, the tracking, bulk filtering, data compression, air traffic control and signal processing.

Correspondingly, for the *Associative Processors' menu*, bearing in mind the cost-factor constraint, the applications include resource allocation, virtual memory mechanism, interrupt processing, protection mechanisms and scheduling.

Of course, there are many other applications where the Parallel Processors are probably cost-effective and in which the Associative Processors would be perhaps more efficient if no constraints were imposed; these applications are, the 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.

Finally, recapitulating, we must mention that Slotnick [*SLOT67*] and Fuller [*FULL67*] have compared both classes of the *SIMD* machines, reaching the conclusion that Parallel Processors, as special-purpose systems, had appeared to be, at that date, more useful than the Associative Processors.

I.B.3.1: The Associative Processor Architecture

Basically, the main notion behind the Associative Processors depends on the extensive search capabilities offered by the associative memories, which are mainly efficient for non-numerical applications, e.g. processing of digitized pictures, radar signal tracking, weather prediction computations, etc.

Although this class of computers was the least favourable, due to the fact that, it was not economical to construct large capacity memories for just non-numerical applications, however, many experimental models were built and a survey of these associative memories and their implementations was produced by A.A. Hanlon [*HANL66*] in 1966.

The pioneer scientists who developed the first associative memory, were Slade and McMahon [*SLAD56*] in 1956, by using *cryotrons*. Many other

different components have been used since then, in constructing such types of memories, such as: *tunnel diodes* (see Fuller [*FULL60*] and Nissim [*NISS63*]), *evaporated organic diode arrays* (see Lewin, Beelitz, and Rajchman [*LEWI63*]), *magnetic cores* (see Fuller [*FULL60*], McDermid, Peterson [*MCDE61*], Kiseda, Peterson, Seelback, Teig [*KISE61*], Hunt, Snider, Suprise, Boyd [*HUNT64*], Younker, Heckler, Masher, Yarbourough [*YOUN64*], Apicella, Franks, [*APIC65*], Goodyear Aerospace Corp. [*GOOD68*]), *plated wires* (see Fuller [*FULL60*], Ewing, Davies [*EWIN64*]), *semiconductors* (see Lee [*LEEE63*]), *transfluxors* (see Lussier, Schneider [*LUSS63*]), *biax cores* (see McAteer, Capobiano, Koppel [*MCAT64*]), *laminated ferrites* (see Wolff [*WOLF63*]), *magnetic films* (see Raffel, Crowther [*RAFF64*]), *solenoid arrays* (see Pick [*PICK64*]), *bicore thin-film sandwiches* (see Fuller [*FULL60*]), *multi-aperture logic elements* (see Tuttle [*TUTT63*]), and *integrated circuits* (see Couranz, Gerhardt, Young [*COUR74*]).

The pioneers who first built an associative processor were Behnke and Rosenberger [*BEHN63*] in 1963, using *cryotrons* and disregarding the existing constraining factors, such as the high implementation cost, the half-select noise limiting the length of the words, as well as the interrogation drive problems limiting the number of words.

The evolution of *LSI* technology has broadened the bounds of the associative memories which were, in early years, about 1k words of length up to 100 bits; then *PEPE*-'Parallel Element Processing Ensemble' (see Crane, Gilmartin, Huttenhoff, Rux, Shively [*CRAN72*], Wilson [*WILS72*], Cornell [*CORN72*], Evensen, Troy [*EVEN73*], Dingeldine, Martin, Patterson [*DING73*], Vick, Merwin [*VICK73*]), *STARAN* (see Rudolph [*RUDO72*], Batcher [*BATC72*,*BATC74a*], Davis [*DAVI74*]) and *OMEN*-'Orthogonal Mini EmbedmeNt' (see Higbie [*HIGB72*]) associative systems were developed.

The designing idea behind *PEPE* was a number of processing elements, each having a simple 1k×32-bit *RAM*[†], called the *element memory*, which could be shared, on a cycle-stealing basis, by the arithmetic unit, the correlation unit and the associative output unit, in order to perform associative processing.

On the other hand, the *orthogonal* computer concept (see Shooman [*SHO060*,*SHO070*]), which will be discussed later in this *Chapter*, formed the basis of the *STARAN* and the *OMEN* Associative Processors, which were commercially implemented, by the Goodyear Aerospace Corporation, and Sanders Associates, respectively.

The *OMEN* computer utilized a *DEC PDP-11* conventional computer for the Horizontal Arithmetic Unit (*HAU*) and an array of 64 identical processing elements for the associative Vertical Arithmetic Unit (*VAU*) which operated on byte slices, rather than bit slices, under the control of masks.

The idea of constructing the *STARAN* Associative Processor was conceived in 1962, completed in 1972 and by 1976 four systems had been sold. The typical *STARAN* system consisted of a control system, four array modules, each with 256 one-bit processing elements for the associative *vertical* arithmetic unit, and a 256-word×256-bit of *MAM*[‡] to facilitate both the associative processing, by allowing bit-slice accessing, and the I/O procedure by allowing a word-slice accessing. The range of the total storage was between 64K bits and 64M bits, controlled by a sequential *PDP-11* for the conventional *horizontal* arithmetic unit. The evolution of the Large Scale Integrated electronic technology will undoubtedly renew the interest in this class of computers. Many scientists are anticipating that the extensive use of these processors will greatly enhance the *special-*

[†] *Random-Access Memory*.

[‡] *Multidimensional-Access Memory*.

purpose and *general-purpose* computer systems performance. Already a modified model of the commercial *STARAN* computer, with a storage capacity of several hundred million bits, has been developed. Also, various other experimental designs, e.g. the 'Associative Linear Array Processor' (*ALAP*), developed by Finnila, at Hughes Aircraft Co. [*FINN77*], and the 'Extended Content Addressed Memory' (*ECAM*), developed by Anderson and Kain [*ANDE76*] at Honeywell, were promising total system capacities of about one billion bits.

Finally, we emphasize the two main properties which characterize this class of the Associative Processors: a) The single instruction propagated by the central control unit, which can be executed during an arithmetic or logic data transformation over many sets of arguments; and b) it is the retrieval of the operands stored in the associative memory by using the *content* or part of it, of the data, instead of the *address* of the location storing them, which contributes to the superiority, from the higher processing rate point of view, when compared with the ordinary sequential systems. Consequently, problems like the weather forecasting, the radar signal tracking or the *handling* of large databases, demanding heavy and high time-consuming computing, are tackled faster and easier.

In *Figure (I.B.3.1-f1)*, we can see the general outline of such an architecture; due to the impact that the associative memories have on this class of systems, it is possible for these processors to be categorized depending on the organization of the associative memories.

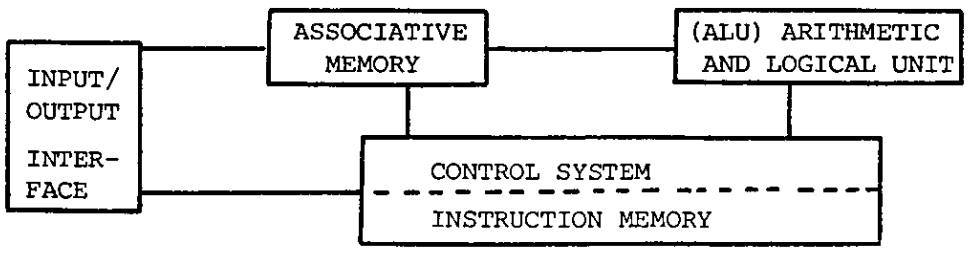


Figure I.B.3.1-f1: The General Scheme of an Associative Processor Architecture.

I.B.3.1.1: THE ASSOCIATIVE MEMORY ORGANIZATION

Many names have been given to the associative memories, such as: *catalog memory* (see Slade and McMahon [*SLAD56*]), *data-addressed memory* (see Newhouse and Fruin [*NEWH62*]), *content-addressed memory* (see Lewin [*LEWI62*]), *parallel search memory* (see Falkoff [*FALK62*]), *search memory* (see Joseph and Kaplan [*JOSE62*], Kaplan [*KAPL63*], Gall [*GALL64*]), *search associative memory* (see Pick [*PICK63*]), *content-addressable memory* (see Estrin and Fuller [*ESTR63*]), *distributed logic memory* (see Lee and Paull [*LEEP63*]), *associative pushdown memory* (see Derickson [*DERI68*]), and *multi-access associative memory* (see Natarajan and Thomas [*NATA69*]).

Disregarding the great number of assigned synonyns to these memories, the main and most significant property of them is the *content-based* retrieval capability of the stored data, instead of their explicit location (address). This can be achieved by either purely *software* techniques, e.g. list structures, hash addressing, etc., or by using *hardware* methods and therefore depending on the basic memory element, the so-called *bit-cell*, which can retain a single bit of information in order to compare it with the interrogating information.

The memory search is based on a comparison procedure, performed, either parallel-by-bit (word-parallel or word-serial), or serial-by-bit, between a pre-defined *search-key* word, and the other words in memory, through the available logic circuitry and the interrogating bit drives.

A *word-match* tag network is used to flag the multiple-matched contents which at the end of the searching procedure can be retrieved through a single instruction. More explanatorily, in order to present more practically the above, we shall make use of the following example: Let us suppose that we have the personnel file of a University department

and we wish to locate these employees with a salary of more than £650 per month and less or equal to £1000 per month, i.e. £650<x≤£1000. The comparison operations that we shall use are *greater than* and *not greater than*.

First of all we must set two different *search-key* words each with the corresponding information to be interrogated; also a mask has to be used for the *search-key* words, thus avoiding a thorough search of all fields in the corresponding files. At the end of the search-key there is a bit-indicator which can take the values of *one* or *zero*, according to the circumstances, thus showing a match or mismatch. Each of the above searches can be performed in parallel. *Figure (I.B.3.1.1-f1)* shows this operation of the associative memory.

Initially, for the two different searches the indicator field of each word, in the associative memory, is set to *zero*; the first search-key word is loaded with the salary figure (£650) and the bit-indicator set to *zero*, for comparison, using the logic operation *greater than*. After the first search, all the matched words will have set the field indicator to *one*, to memorize these words. The second search-key word will have been loaded with the salary figure (£1000) and the bit-indicator will be set to *one*, for the logic operation *not greater than*. The final result, which will be signalling from the indicator field, will show the salaries satisfying both the above conditions, and consequently these data can be printed out through the output circuit.

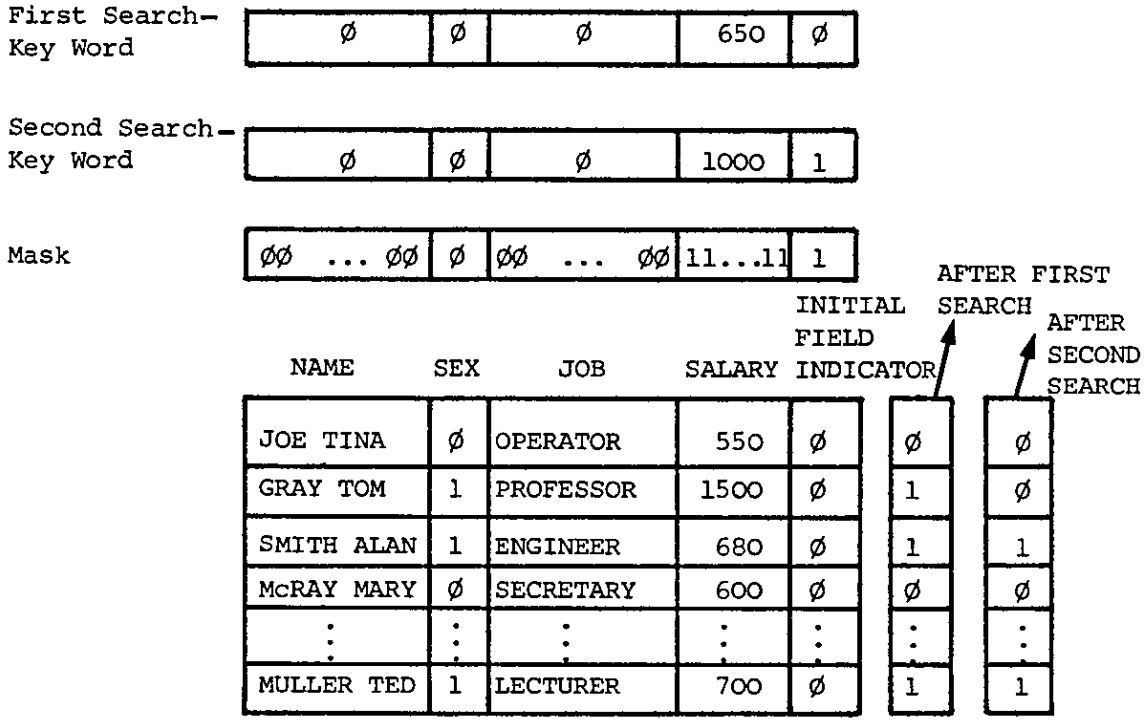


Figure I.B.3.1.1-f1: An Example of the Operation of an Associative Memory.

I.B.3.1.2: Architectural Taxonomy Of The Associative Processors

The Associative Processors, from the hardware structure point of view, form a subclass of the general category of *SIMD* architectures. In the *SIMD* systems we have a single instruction being propagated, instructing the processing elements of the organization to operate on different arguments. Similarly, the Associative Processors, in addition to the fact that they satisfy the characteristic property of the associative memory when retrieving the data, they can perform both arithmetic and logic data transformation operations, over many sets of arguments under a single instruction from the control unit.

The comparison process followed by the associative memory for the retrieval of the data forms the basis for a classification into five

architectural Associative Processor categories: The *fully parallel*, the *bit-serial*, the *word-serial*, the *block-oriented*, and the *highly parallel* Associative Processors, the first two being the most widely known and important categories.

In the following paragraphs we shall briefly refer to each of these categories separately, presenting their main hardware features and examples of commercially implemented systems.

I.B.3.1.2.i: FULLY PARALLEL ASSOCIATIVE PROCESSORS

This class of computers can be further distinguished into the *word-organized* and the *distributed logic* types of Associative Processors.

In the first subclass, the logical functions for comparison are related to each bit-cell of every word in the memory and consequently we obtain a parallel-by-word and parallel-by-bit comparison processing, with the logical decision known at the output of every word. Although this is the fastest version of the Associative Processors, however, the high hardware complexity involved in each bit-cell containing a separate logic circuitry, justifies the fact that all implementations of this type of computers have remainded at the experimental level. In *Figure (I.B.3.1.2.i-f1)* we can see the general structure of such an architecture.

The other type of this category has the logical function for comparison related to each character-cell (for a fixed number of bits) or a group of character-cells. In *Figure (I.B.3.1.2.i-f2)* we can see the general structure of such an architecture.

Lee [*LEEC62*] in 1962 was the pioneer to propose such a system, but the best known developed system is *PEPE*, originally developed by Bell Laboratories for the U.S. Army Advanced Ballistic Missile Defense Agency.

A few years later, in 1965, Crane and Githens [*CRAN65*] introduced an extension to Lee's system, a two-dimensional distributed logic memory, utilizing a large number of identical processing elements, thus achieving a higher executional performance for arithmetic operations.

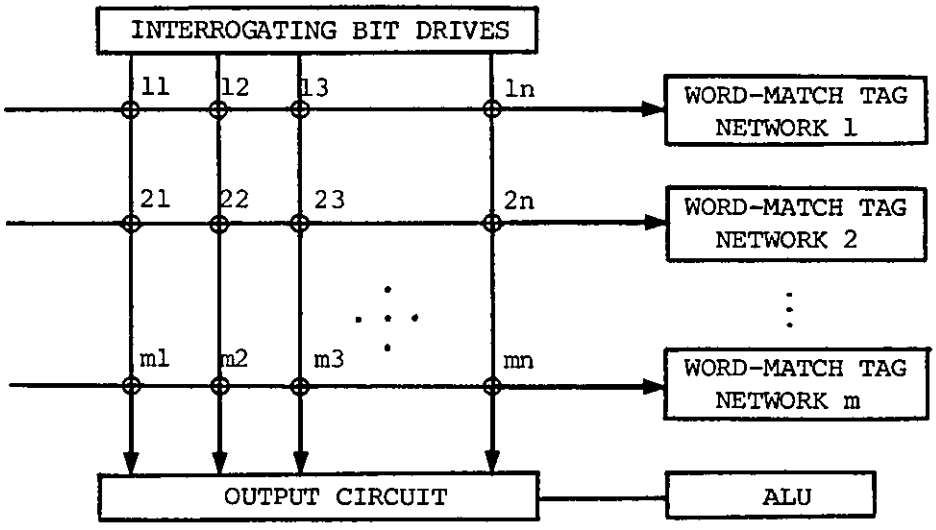


Figure I.B.3.1.2.i-f1: The General Structure of a Fully Parallel, Word-Organized Associative Memory and *ALU* (each crosspoint represents a bit-cell of the memory).

Of course many complications have been encountered in implementing such types of computers, but step after step they have been researched carefully and solutions offered; for example, Lipovski [*LIPO70*] in 1970 designed the 'Tree Channel Processor' (*TCP*) which is an interconnection scheme attempting to solve the problems which occurred by the requirements of the *DLMs*[†] for high-speed I/O and ability to segment the processor to execute subprograms.

Finally, another extension to the *DLM* was the 'Association Storing Processor' (*ASP*)- see Savitt [*SAVI66*], which was an attempt for even faster processing resulted in two special-purpose machine designs, the *item-oriented ASM* and the *phrase-oriented ASM*.

[†] *Distributed Logic Memory*.

The general *ASP* consisted of much more than a processor and the main principle of its design was that the hardware was especially designed to support the language to be used on it. The main difference was that the *phrase-oriented* version, instead of identifying single items, was capable of storing, searching and replacing phrases, i.e., sequences of relations (substructures of the data structure).

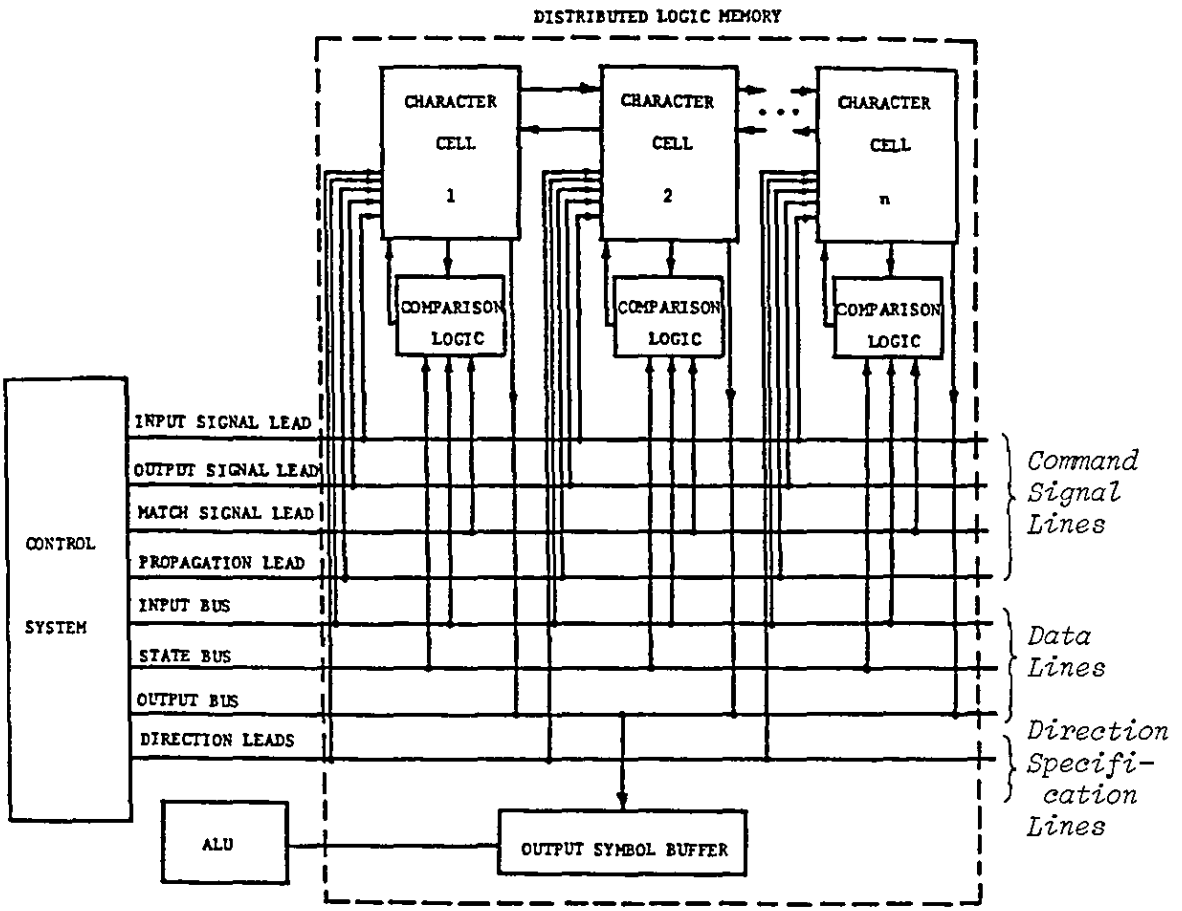


Figure I.B.3.1.2.i-f2: The General Structure of a Fully Parallel Distributed Logic Associative Processor (as proposed by Lee).

I.B.3.1.2.ii: BIT-SERIAL ASSOCIATIVE PROCESSORS

The expensive logic required for each memory bit in the previous class of Associative Processors, as well as the communication problems between them, resulted in the appearance of the *bit-serial* associative processors. Shooman's concept on using *vertical* data in parallel processing formed the basis of this class, since, comparatively, the number of bits, in each word to be processed, is smaller than the total number of words.

Ewing and Davies [*EWIN64*] in 1964, first proposed the design logic for such a computer; in *Figure (I.B.3.1.2.ii-f1)* we can see the outline of a *bit-serial* associative memory, where each intersection of a bit-line and a word-line is of one bit-storage. The operation takes place at *bit-slices*, selected by the bit-column select logic, and the associative processing is obtained through the *word logic* associated with each word-line. This is identical for all words and consists of a sense amplifier, storage flip-flops, a write amplifier and a control logic. In contrast to a distributed logic Associative Processor, such a processor can be considered as an *external-logic* associative processor.

Bit-serial Associative Processors have been implemented through the use of 2½D core search memory (see Harding and Rolund [*HARD68*], Stone [*STON68*]). Goodyear Aerospace Corporation developed one of the most widely known bit-serial associative processor, the *STARAN* computer, consisted of an optional number - up to 32 - of associative array modules. The most significant part in this system is the *permutation network* or otherwise *flip* or *interconnection network*, through which the communication between the processing elements and the memory modules is obtained. In *Figure (I.B.3.1.2.ii-f2)* we can see the operational concept of such a type of computer; after the propagation of the single instruction, the processing

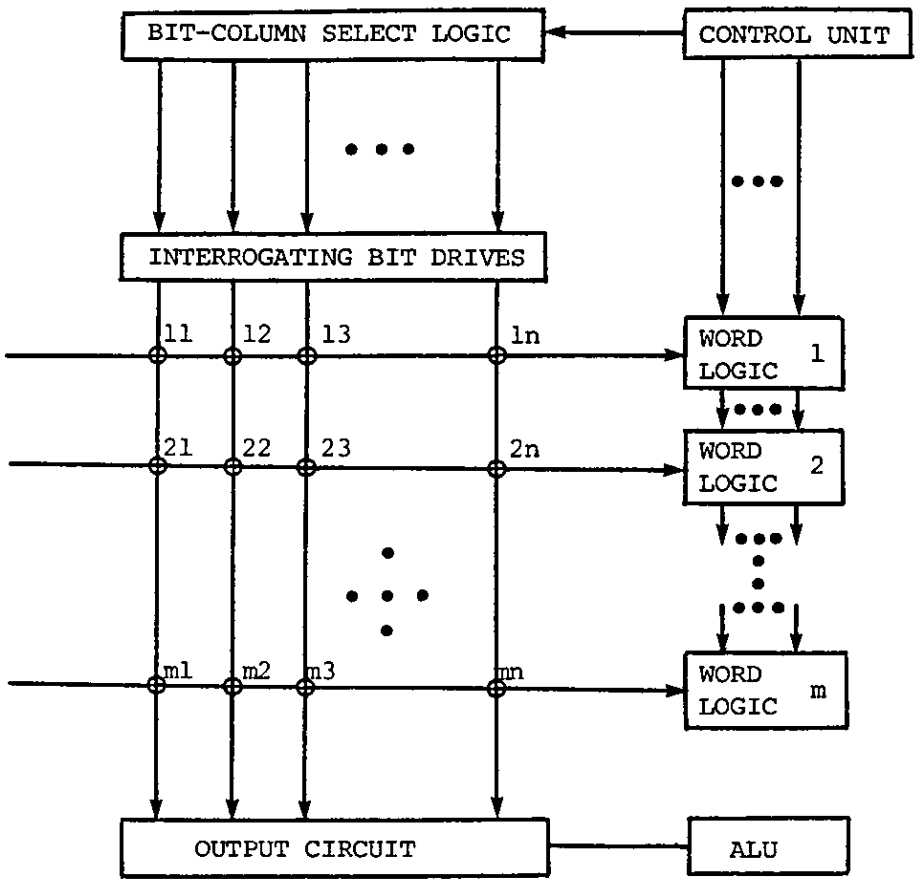


Figure I.B.3.1.2.ii-f1: The Outline of a Bit-Serial Associative Memory and the *ALU.*

elements corresponding to those words which satisfied the search criteria, start to operate on each word simultaneously, until the seeking data argument is finally located.

Occasionally, some heavy demanding data processing problems, as the weather forecasting or the air traffic control systems, can be tackled much faster by a *hybrid* system composed of an Associative Processor to operate on tasks suited for it in parallel and a conventional sequential processor to handle tasks which must be processed in a single sequential data stream.

Concluding, we mention some other examples of bit-serial Associative Processors, like the *OMEN* computers - developed by Sanders Associates, the

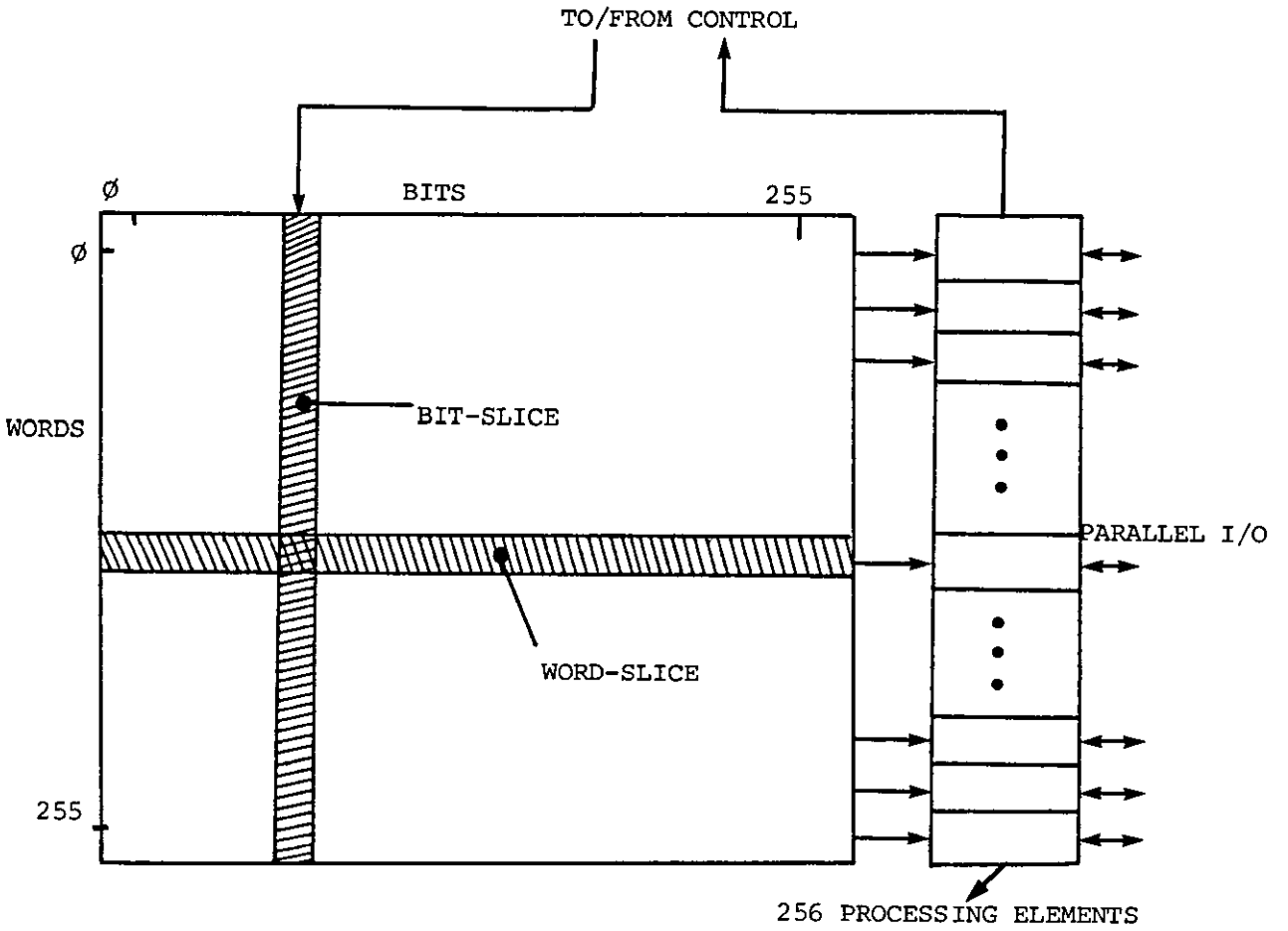


Figure I.B.3.1.2.ii-f2: The Operational Concept of a *STARAN* Associative Array Module with a 256-Words× 256-Bits Memory.

hybrid Associative Processor - developed by Hughes Aircraft Co. (see Love [*LOVE73*]), the 'Raytheon Associative/Array Processor' (*RAP*) - (see Couranz, Gerhardt, and Young [*COUR74*]), the 'Associative Linear Array Processor' (*ALAP*) - (see Finnila and Love [*FINN77*]),which can also be considered as a distributed-logic bit-serial Associative Processor, and the 'Extended Content Addressed Memory' (*ECAM*), - (see Anderson and Kain [*ANDE76*]).

I.B.3.1.2.iii: Word-Serial Associative Processors

This subclass of Associative Processors has not been commercially implemented due to the fact that they have not promised high executional speeds. Such a type of computer, simply represents a hardware implementation of a program loop for searching the data; on this fact depends the improved efficiency compared to conventional sequential computers, since

only one instruction is required for the word search, each time, and consequently the instruction decoding time is minimized.

The first to present an experimental model of this type of computer were Crofut and Sottile [*CROF66*], in 1966. The basic constituent of that model was a word-serial associative memory with operational characteristics very similar to that of a drum or disk; the memory utilized n ultrasonic digital delay lines, operating at 100 MHz, with 10 μsec delay time, where n is the number of bits/word.

This memory was under the control of a *rewrite control logic* which checked the informational traffic through the delay line system and would allow either the same data to circulate or new data to flow into it. Each bit/word occupied a delay line and a synchronous propagation of all the bits of the word through the delay lines took place, thus making feasible the individual interrogation and updating of every word at the exit of the delay line system. The synchronizing clock pulses, in order to advance the address counter, were provided by a stable oscillator (*Stalo*). The hardware outline of such a type of computer can be seen in *Figure (I.B.3.1.2.iii-f1)*.

I.B.3.1.2.iv: Block-Oriented Associative Processors

This subclass of the Associative Processors, can be seen as a compromise between the low-speed word-serial and the expensive bit-serial Associative Processors, offering low-cost associative processing which is very efficient in the cases where a large data argument storage and retrieval system are required.

Various models, of this type of computer, have been developed, the pioneer scientist, who presented the first *block-oriented* Associative

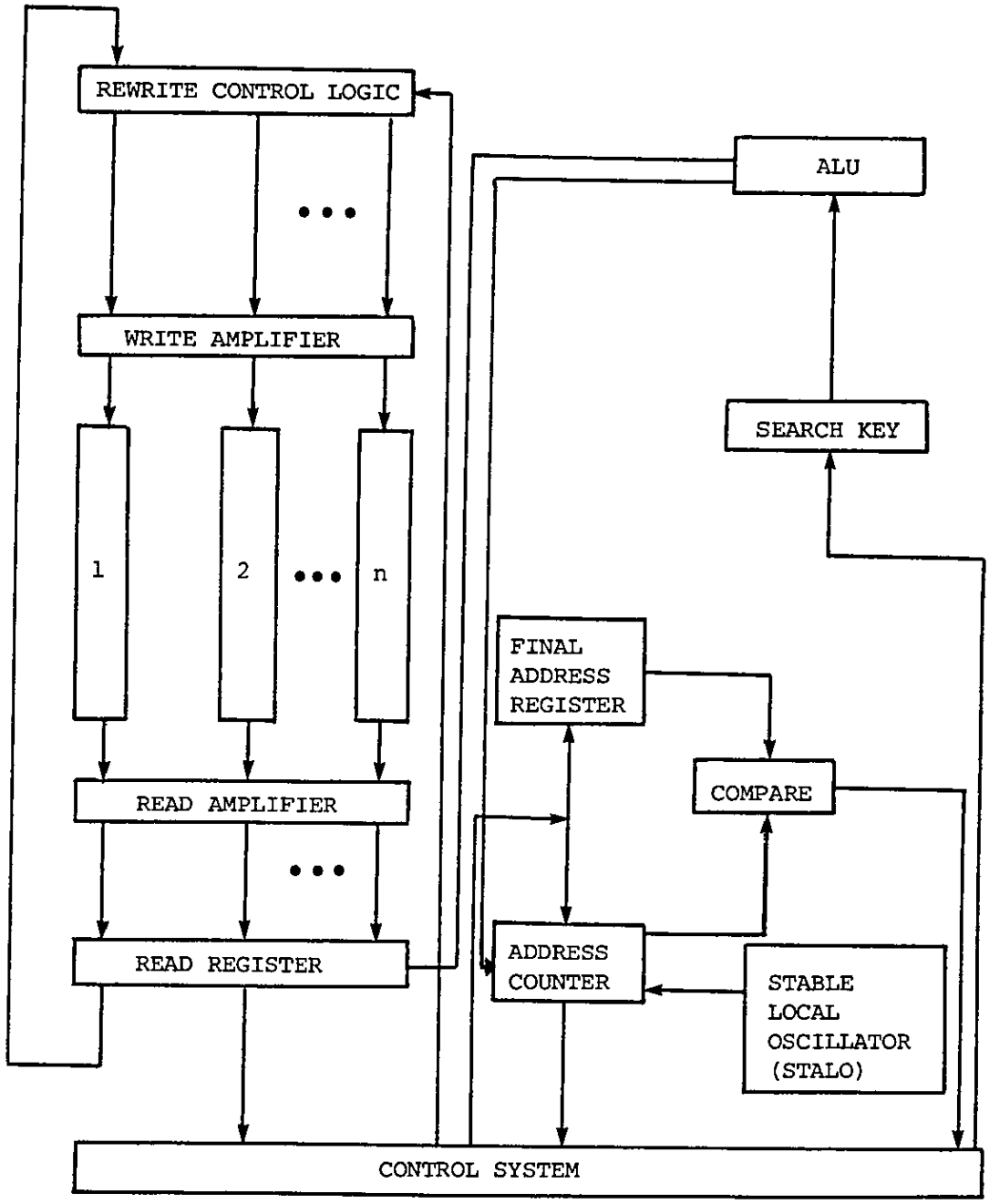


Figure I.B.3.1.2.iii-f1: The Hardware Interconnections of a Word-Serial Associative Processor.

Processor called *RAPID* ('Rotating Associative Processor for Information Dissemination'), being Parhami [*PARH72*] in 1972. This computer, whose associative memory can be seen in *Figure (I.B.3.1.2.iv-f1)*, was based on both, Slotnick's [*SLOT70*] and Parker's [*PARK71*] concept of using *logic-per-track*

devices, i.e. disk memories having a head and some logic on every track, and on Lee's distributed logic memory to store and retrieve information.

After Parhami, Minsky [*MINS72*] in 1972, among other scientists who presented various modified versions of the above idea, utilized the *delay time* spent to locate a given address, for content information retrieval. He introduced the term *partially associative memory* using rotating drum or disk memories controlled by a *Controller* Processor to store the information according to their basic structure, i.e. *name part* and *data part*, as well as the operational characteristics, i.e. *instructions* and *predicates*.

In a recapitulation of Associative Processor systems, we mention that they are very efficient for high-speed parallel information processing and from the above categories, based on the associative memory organization, the first two are the most widely known commercially.

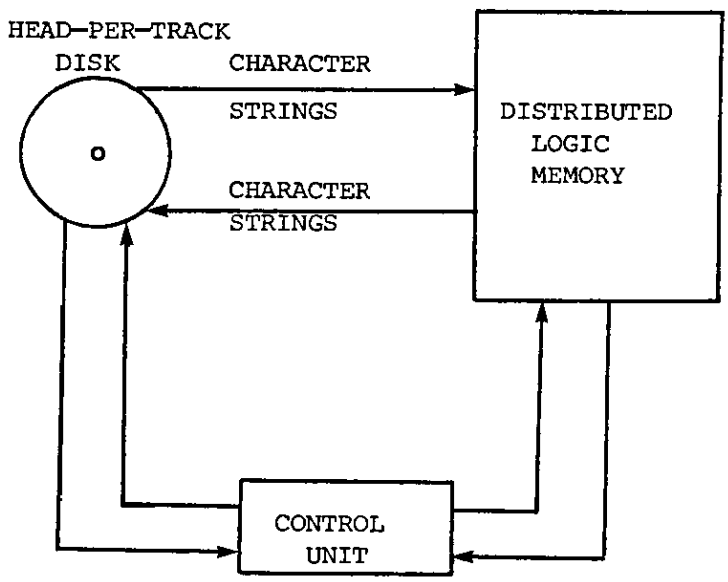


Figure I.B.3.1.2.iv-f1: The Associative Memory of *RAPID*.

I.B.3.1.2.v: HIGHLY PARALLEL ASSOCIATIVE PROCESSORS

In the last subclass of the Associative Processors we shall refer to some architectures which, apart from their associative features, possess either multiple control units or an extensive control built inside the memory array.

An example of this type of computer is the *ACAM* ('Augmented Content-Addressed Memory') system, designed by Kautz [*KAUT71*] in 1971; in this system, each storage cell incorporates many functions thus providing an extensive arithmetic capability, which is traded-off against an excess amount of complexity compared to the conventional Associative Processors, e.g. there are 40 NOR gates/cell instead of the 10 gates/cell in a simple Associative Processor.

Thus *ACAM*, is an $(m \times n)$ array of cells, where each cell consists of many different functions; in each cell there are three registers dedicated to hold the programmable values c_i, b_i, a_i in a column fashion which are used to define the behaviour of the array. The c values, which are constant along a column, are used as bit masks to select the required operation columns, whereas a *word-based* logic pairs the ab values, which are constant over each word; the latter values are used to select the type of function, uniquely determined by their intersection with the c column, to be performed on the specific cell. In other words, this means that different operations can be simultaneously performed on different words. Of course all this flexibilityhas caused an increased complexity according to which each cell has 8 terminals and the $(m \times n)$ array has $4.(m+n)$ edge terminals.

Finally, for special purpose applications, i.e. bulk filtering of radar data, another type of Associative Processor was designed by Schmitz, et al [*SCHM72*] in 1972, attempting to utilize the *dormant* Processing

Elements (*PEs*) occurring in a conventional Associative Processor during the instruction execution phase.

This system comprises *m* control units and *n PEs* of various degrees of complexity. The main constituent in this architecture is the *ACS* ('Associative Control Switch') which acts as an extended search-results register, i.e. according to the result(s) of an operation(s) performed in a *PE*, the *ACM* selects the suitable control unit for this *PE* or leaves it inactive; consequently different parts of the architecture can operate simultaneously on different instruction streams.

In conclusion, we mention that other researchers in the past (see Seeber and Lindquist [*SEEB63*]) have proposed other schemes in which instead of using simple *PEs* have assumed whole Associative Processors. However, most of these have remained in the experimental stage due to the difficulties of constructing large associative memories.

I.B.3.2: PARALLEL PROCESSOR ARCHITECTURES

Under the Parallel Processor architecture category, according to the presented genealogy of the *SIMD* computers, we shall examine the characteristics of the so-called actual *Array* or *Parallel* Processors, relatively going into some more details about the *ILLIAC IV* and especially the *ICL DAP* computers.

This is mostly because, the influence of the former computer had a profound effect on this class of systems, although it may be considered as a failure in that it cost four times the original contract figure, and its actual performance did not come even within a factor of 10 of its expected performance; on the other hand, the latter machine is accessible from Loughborough University through a *modem*. Finally and briefly, we shall

refer to Shooman's concepts for the Orthogonal Processor classified in this category but mainly offering the basis for constructing Associative Processors.

One of the first theoretical descriptions of such a parallel processing system is due to Richardson, L.F. [*RICH22*] in 1922, who attempting to do a weather forecasting entirely by hand spent an excessive amount of time. This work led him to propose an operational *forecasting factory* consisting of an *army* of 64,000 'computers' - each probably being, according to the age of computers then, a person using a present day desk calculator - each of which would be assigned to the atmosphere column above a predetermined location on the earth's surface, and all the calculations of them would be coordinated by a general *director*.

All the systems in this class can be roughly identified of consisting of some major components, harnessed together in a variety of different ways: 1) A number of identical processors, centrally driven and synchronized, adjusted in 2) various forms of communication networks; 3) a number of memory banks, not necessarily equal to the number of processors, 4) some form of local control, and finally 5) some form of global control. The general hardware model of an Array or Parallel computer system, comprising p processors and parallel memories, can be seen in *Figure (I.B.3.2-f1)*.

The control unit, which is usually a computer itself, called the *host* computer, provides the *common* instruction stream, and a *private* memory is associated with each processor, which supplies it with the data stream, on which it executes the propagated instructions in a *synchronous* fashion way. These systems are capable of achieving speed-ups proportional to the number of processors they contain.

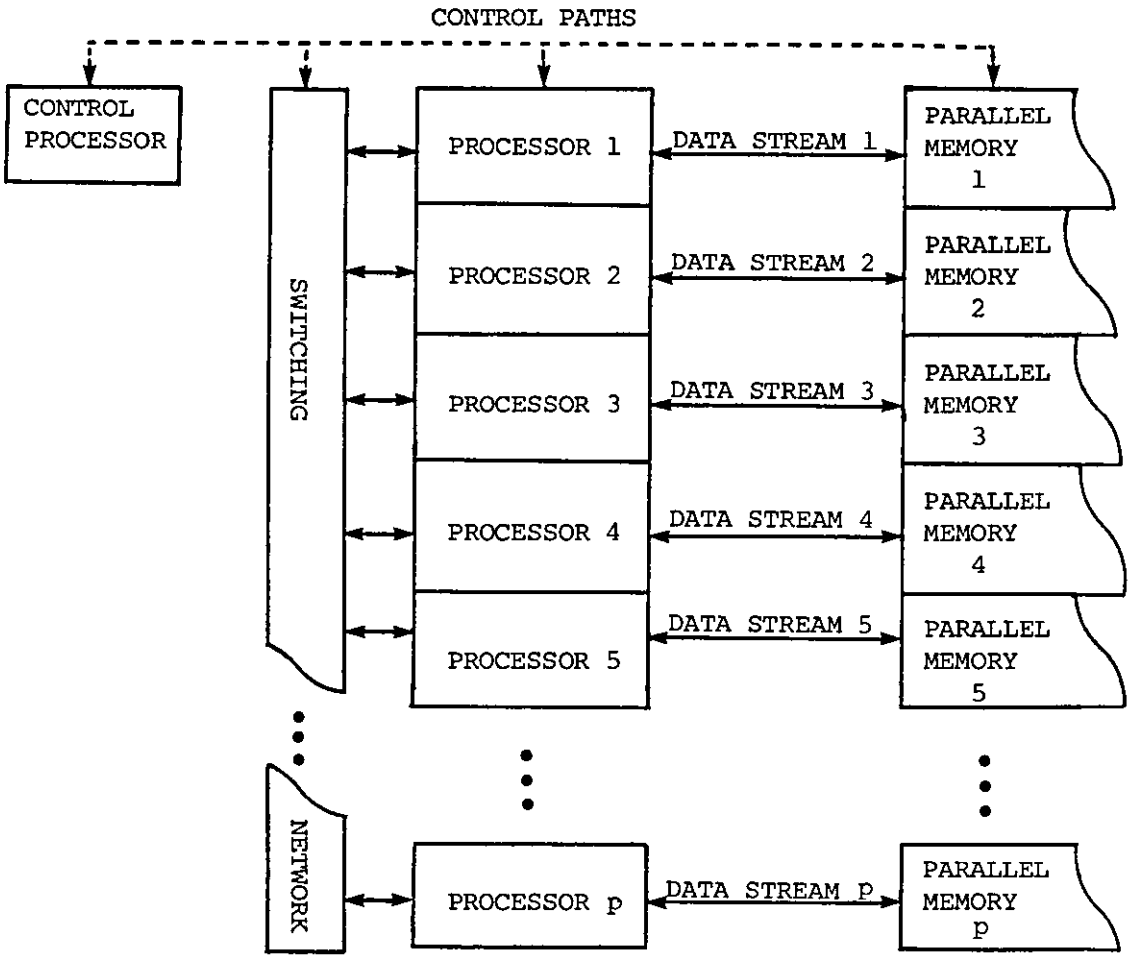


Figure I.B.3.2.-f1: The General Model of an Array or Parallel Computer, having an identical number of Processors and Memory Banks.

Of course there are many difficulties in using these *parallel* memories, which we shall discuss in the following paragraphs, the major one being the accessing conflicts which may arise and consequently slow down the whole system. In fact, if a *speed balance* has been achieved amongst memories and processors and if parallel processing cannot proceed until every required element has been fetched from the corresponding memories, then the system is slowed down by a factor equal to the maximum number of conflicts in any memory module.

I.B.3.2.1: The Utilization Of Parallel Memories

Any memory system that contains a number of separately addressable memory modules is called a *multimemory* or *parallel memory* system. More specifically, if in a m-multimemory system the successive addresses are assigned across the memories, modulo m, it will be called an *interleaved* memory system. The terms *phased* and *interlaced* memories sometimes are used to refer to interleaved memories and at other times refer to designs in which each of the parallel memory modules is cycled on a different minor clock cycle.

Generally, the *parallel memories* are of high significance for the Parallel or Pipelined computers, because at the word-level parallelism, i.e. each memory produces one word/mc[†], they offer the means to maintain a balance between the unevenly increasing speeds, according to the technological evolution, of the processors and the corresponding memory banks.

Many techniques have been developed in an attempt to solve the most serious problem of conflicts occurring in the use of these memories during the accessing of the stored data. Since multidimensional arrays are used frequently in programs, we consider a simple example of storing a two-dimensional ($m \times n$) array, where $m=n=4$, see *Figure (I.B.3.2.1-f1)*. The storage has taken place columnwise, thus the data is suitable for *row* or *diagonal* access, this however presents conflict problems when accessing a *column*, and therefore it is necessary to cycle the memories that many times, as the number of elements in a column.

Various alternative storing techniques have been developed for specific problems; for example, by *skewing* the data along the memories,

---

[†] *Memory Cycle.*

see *Figure (I.B.3.2.1-f2)*, the conflicts in retrieving a *row* or *column* of data will be avoided, but not for the *diagonals*. In fact, there is only one way to store an $(n \times n)$ matrix in n memories, when n is even, so that arbitrary rows, columns and diagonals can be fetched without conflicts, this is if we consider more than n memories.

MEMORY UNITS ⟶	Ø	1	2	3
	x_{00}	x_{01}	x_{02}	x_{03}
	x_{10}	x_{11}	x_{12}	x_{13}
	x_{20}	x_{21}	x_{22}	x_{23}
	x_{30}	x_{31}	x_{32}	x_{33}

Figure I.B.3.2.1-f1: A typical straight storage of a Two-Dimensional (4×4) Array into 4 Memory Units.

MEMORY UNITS ⟶	Ø	1	2	3
	x_{00}	x_{01}	x_{02}	x_{03}
	x_{13}	x_{10}	x_{11}	x_{12}
	x_{22}	x_{23}	x_{20}	x_{21}
	x_{31}	x_{32}	x_{33}	x_{30}

Figure I.B.3.2.1-f2: A *Skewed* storage into 4 Memory Units allowing access to Rows and Columns of a Two-Dimensional (4×4) Array.

Of course, the *skewed* storage is not a simple scheme, since it requires each memory module to be provided with an independent indexing mechanism allowing access to a different relative location of them. Thus, this is one way to manipulate the behaviour of the parallel memories, since other existing computers - see the *Control Data STAR* and the *Texas ASC* - have adopted the *physical array transposition*, mechanism which although it produces a wasted transposition time overhead, offers the capability

to access a pair from the rows, columns and diagonals accordingly.

This is not the only problem that arises in *parallel memories* nor the most important, since there are many others, on which the usefulness of these memories depends, such as, the kind of mechanism that is assumed for retaining unserviced accesses and the kind of data dependencies assumed in the memory access sequence, which have accepted a wide investigation by the researchers in the past 15 years.

I.B.3.2.2: The Interconnection Networks

Assume we have a computer architecture consisting of a number of processing units and memories; the question which arises is, 'What kind of interconnection techniques are employed, with what costs, speeds and priorities?'. The units that are interconnected can be of various types, starting from different components within a processor, i.e. registers, arithmetic units, up to different processors, memories, I/O devices.

We shall not attempt a detailed answer to the above question, but roughly distinguish the types of the networks in use[†] and the different topologies they can be classified into. The imminent complications of using these networks, i.e. the conflicting simultaneous demands on the interconnection network, are settled by the use of a 'Network Control Unit'.

Generally, we can distinguish two types of *interconnection networks*, the *bus* and the *alignment* networks, whereas the latter can be topologically subclassified into *static* and *dynamic* networks. It is not possible to make a rigorous distinction between buses and alignment networks, but basically through the former one only two units can communicate at one

[†] *In general reduced interconnection networks are utilized, since the complete ones are very expensive.*

instant of time, whereas through the latter one a number of units can communicate in parallel. Consequently, it follows that a bus is a slower, less expensive network than an alignment network.

The alignment networks are, not only, used in the *SIMD* computers, but in the Pipelined and Multiprocessor systems as well, interconnecting processors and memories allowing transmission from memory to memory, processor to processor, or back and forth between processors and memories.

The topologically *static* networks are characterized by the required dimensions for layout and consequently, see *Figure (I.B.3.2.2-f1)*, we can have from one-dimensional topologies, e.g. linear array used for some Pipelined architectures, up to hypercube structures.

On the other hand, the topologically *dynamic* networks are distinguished into the *single-stage*, *multistage* and *crossbar* types of networks. The *single-stage* or *recirculating* network is composed of a stage of switching elements cascaded to a link connection pattern; the *multistage* network consists of more than one stage of switching elements, whereas with a *crossbar* switch network we can achieve any one-to-one connection of inputs to outputs and, even more, one-to-many connections for broadcasting, but this switch is quite expensive and consequently not efficient for large systems. In *Figure (I.B.3.2.2-f2)* we can see some examples of the topologically *dynamic* networks.

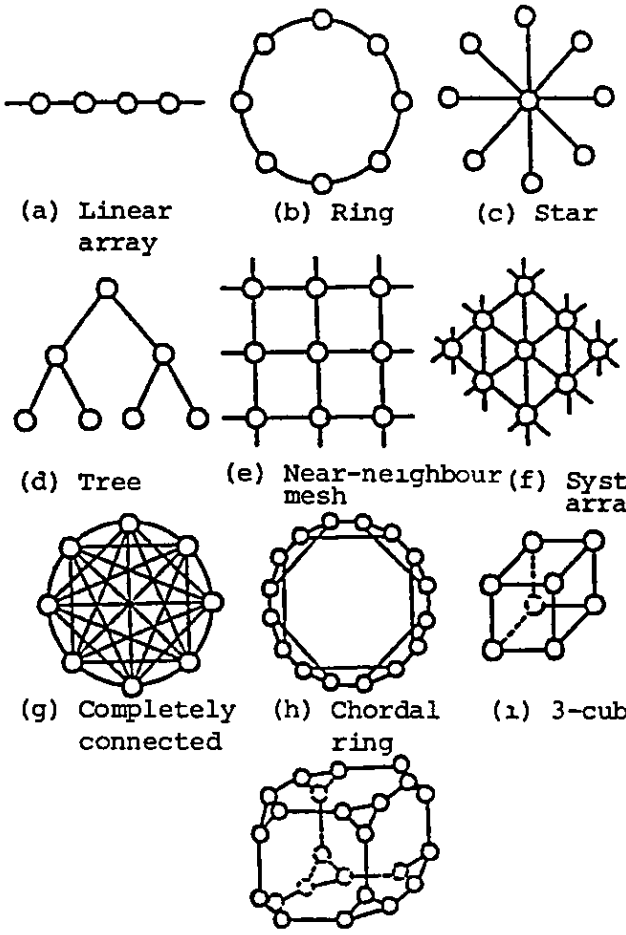


Figure I.B.3.2.2-f1: a) One-Dimensional; b)-f) Two-Dimensional; g)-j) Three-Dimensional Networks.

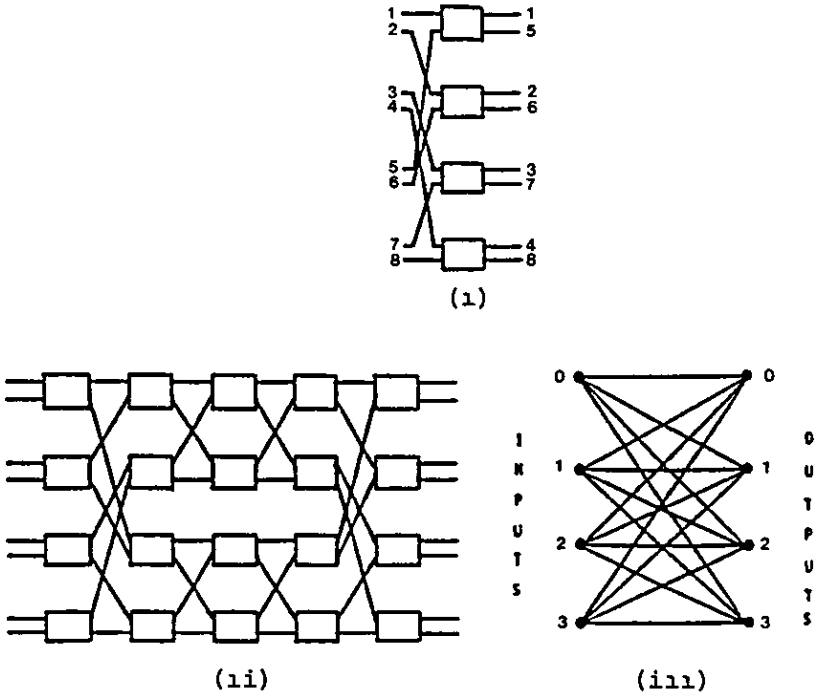


Figure I.B.3.2.2-f2: i) Single-stage, 8×8 Shuffle-Exchange Network; ii) Multistage, 8×8 Benes Network; iii) Crossbar Switch Network.

I.B.3.2.3: IMPLEMENTED PARALLEL PROCESSOR SYSTEMS

In the parallel processor systems area, scientists began, primarily, investigating architectures consisting of arrays of processing elements connected in a four-nearest-neighbour manner. The characteristics which will primarily distinguish one system from another are:

i) The number of processors;

ii) the interconnection paths between the processors;

iii) the parallel memory module sizes;

iv) the complexity of the operations supported by the processors hardware;

and v) the communication paths to other *host* systems.

The Unger's computer, (see [*UNGE58*]),was designed for pattern-recognition processing, mainly dealing with typical line-manipulation functions, as line thinning, extending, doubling, etc; the four-nearest-neighbours connected type of array of *PEs* was capable of allowing conditional jumps to the main control computer by utilizing a test mechanism searching for *zero* values in a designated bit of each *PE* in the array.

The *SOLOMON I*, *SOLOMON II* (the acronym stands for 'Simultaneous Operation Linked Ordinal MOdular Network) and *ILLIAC IV* computers were designed by Slotnick, et al (see [*SLOT62*],[*WEST64*],[*SLOT67*]). *SOLOMON I* was a bit-serial processor and each *PE* contained a serial accumulator; a change of the slow arithmetic unit concept to a 24-bit floating-point unit led to the *SOLOMON II* version of computer.

Summarizing, in general, the *SOLOMON* system, which was a major milestone in the history of parallelism, had *three* principal features:

1) A single control unit controlled a large array of *PEs* interconnected in a four-nearest-neighbour manner, thus providing a moderate coupling

for data exchange;

ii) the central control unit broadcasted the common memory addresses and data to all *PEs* ; and,

iii) each *PE* could perform local tests and so a form of local control was obtainable;

therefore the execution of the common instructions was controlled on a per individual basis scheme.

The *SOLOMON* computer was never built as it was described in [*SLOT62*], but led to the design of two more significant systems, the *ILLIAC IV* (designed in 1966, in the University of Illinois under a contract of the U.S. Dept. of Defense's 'Advanced Research Projects Agency' - *ARPA*) and the *ICL DAP* array of single bit *PEs*.

The *ILLIAC IV* was primarily designed for the solution of Partial Differential Equations and was never built as envisaged, due to the fact that the tendency to attempt to implement such pioneer projects gradually superseded the original objectives for greater efficiency. Actually, only 1/4 of the original configuration was finally built by Burroughs and delivered to NASA Ames Research Center, California, in 1972; but even this architecture can be regarded as a failure, not only from the high cost, but also from the operational point of view, with delays in routing the data to long distances across the array, caused by the limited nearest-neighbour connections between the 64 *PEs* and the 64 corresponding banks of *PE* memory. This problem forced Burroughs to reduce the number of processors and memory banks to 16 and 17, respectively, when they constructed the commercial *BSP* system (see Jensen [*JENS78*], 1978).

The primary configuration of *ILLIAC IV*, see *Figure (I.B.3.2.3-f1)*, comprised four *quadrants*, each consisting of 64 floating-point *PEs*, in a

(8×8) array formation, with a 2048 words of 64-bit thin-film memory, with a 240ns cycle time, assigned to each of them. These quadrants would be interconnected through a highly parallel I/O bus and the 256 coupled *PEs* would be driven by instructions from a common control unit. Also, a 10^9-bit, head-per-track disk, with a 40-ms rotation speed and an effective transfer rate of 10^9 bits/sec, was to be provided as a secondary memory to read from and write to.

Although the maximum processing rate achieved was approximately 50 M flops/s out of the expected 1G flop/s, and the original clock period lengthened from 40ns to 80ns, this computer was the first to use semiconductor memory chips for the main memory, i.e. 256-bit bipolar logic gates, since there was not enough space to use thin-film memories; these gates were packed about 7 per chip (*SSI*), due to the use of the new and faster *Emitter-Coupled Logic* (*ECL*) instead of the *Transistor-Transistor Logic (TTL)*. Also this system used 15-layer circuit boards and computer-aided layout methods to wire them.

The complete system included a Burroughs *B6500* computer to hold most of the software which contained *four* programming languages that could exploit the systems' parallelism, i.e. the Algol-like *TRANQUIL* (see Abel, et al [*ABEL69*]), the Pascal-like *ACTUS* (see Perrott [*PERR78*]), the *GLYPNIR* (see Lawrie, et al [*LAWR75*]), and the *CFD FORTRAN* (see Stevens [*STEV75*]). In conclusion, this computer architecture was too far and ambitiously ahead of its technological time, to be considered as practicable, but undoubtedly was a profound step forward in parallel computer architecture.

The pilot model of the ICL 'Distributed Array Processor' (*DAP*) started in 1974, attempting to achieve a balance between computational power and inexpensive computing construction technology, and finally commissioned in

1976 (see Reddaway [*REDD79*]); conceptually it was very close to the initial *SOLOMON* computer (see [*SLOT62*]) which consisted of a (*32×32*) array of one-bit *PEs* each with 4096 bits of memory performing its arithmetic on 1024 numbers simultaneously in a bit-serial fashion.

However, the design of *DAP* introduced two new features to the *SOLOMON* concept: The *first* one, was a hardware slicing of the array in two orthogonal directions, whereas a number of registers in a 'Master Control Unit' (*MCU*) were capable of alignment with either dimension, by the use of two orthogonal data *highways* threading the rows and columns of the *PEs*. These highways have one bit for each bit in the *MCU* register which terminates in either a row or column of the array, e.g. PE_{ij} possesses a one-bit highway directly to bits i and j of the *MCU* register. The *second* feature was that *DAP* not only emulated a memory module for the *ICL 2900* mainframe computer to which it was attached, but, also it was autonomously capable of processing data in a highly parallel manner.

The organization on a bit word basis rather than a 64-bit word constitutes the main difference between the *PEs* of this system and those of the *ILLIAC's IV*.

The first implemented machine of this type was installed at Queen Mary College, London, in 1980, comprising a (*64×64*) matrix of *PEs* with nearest-neighbour connections, each capable to operate independently on its own local store; consequently the *DAP* store was the total sum of 4096 local stores. The major components of the *DAP* system can be seen in *Figure (I.B.3.2.3-f2)*.

Another pioneer feature of the *DAP's* design is that the *PE* logic is placed, along with the memory to which it belongs, on the same circuit board, which helps to avoid the von Neumann's machine *bottlenecks* due to

the separation of these two in different components. Furthermore, using the *VLSI* technology the *PE(s)* and corresponding memory(s) could be mounted on the same chip, which means that even larger matrices of *PEs* could be constructed, e.g. a *(256×256) DAP*.

Each *PE* contains 3-one-bit registers *A,Q,C*, two multiplexers and a one-bit full *adder* - arithmetic unit to perform the arithmetic. The *A* register provides a programmable control over the *PE's* actions, the *Q* register is an *accumulator* and the *C* register a *carry* store; the full adder adds *Q,C* and the input to the *PE* producing the *sum* and *carry* outputs, storing them in the *Q* and *C* registers, respectively.

Finally, a parallel FORTRAN based language, called *DAP FORTRAN*, has been developed to take advantage of the machine's superior processing power. In *Chapter II* we refer to the programming concepts and performance of this, special, SIMD system, since in *Chapter IV* we mention some results obtained in the Computer Studies Department from it.

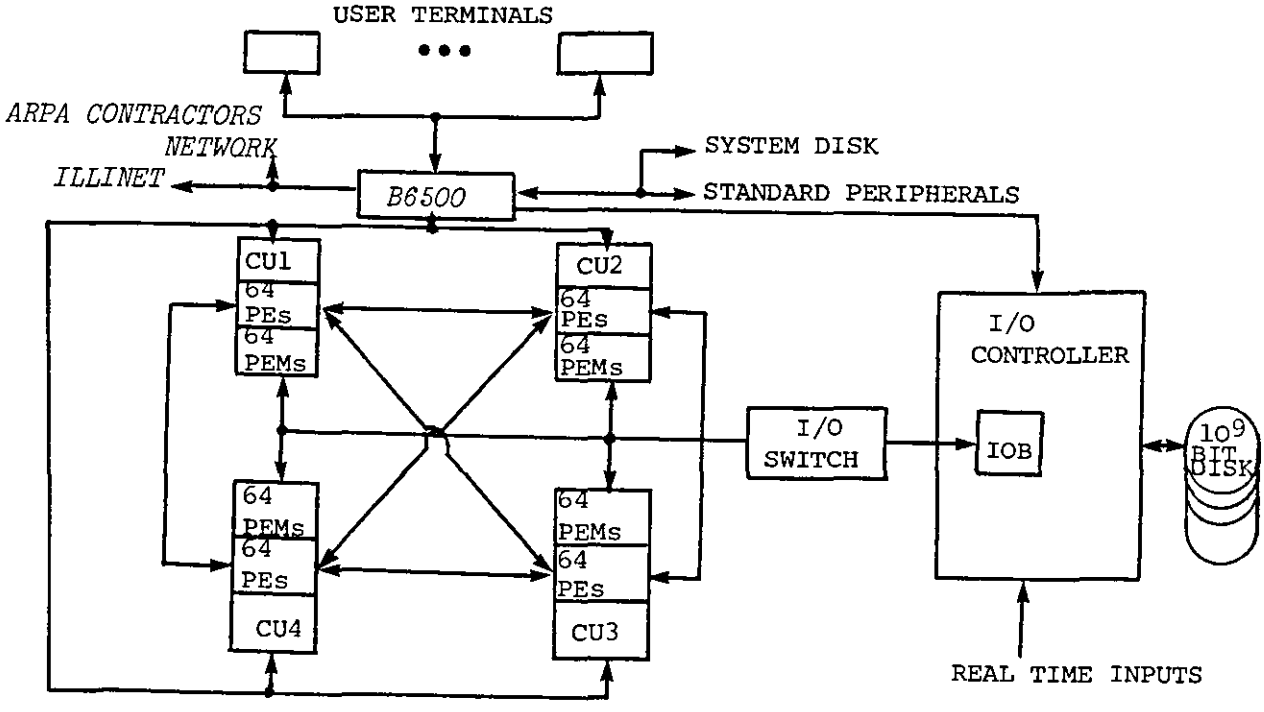


Figure I.B.3.2.3-f1: The *ILLIAC IV* System Configuration.

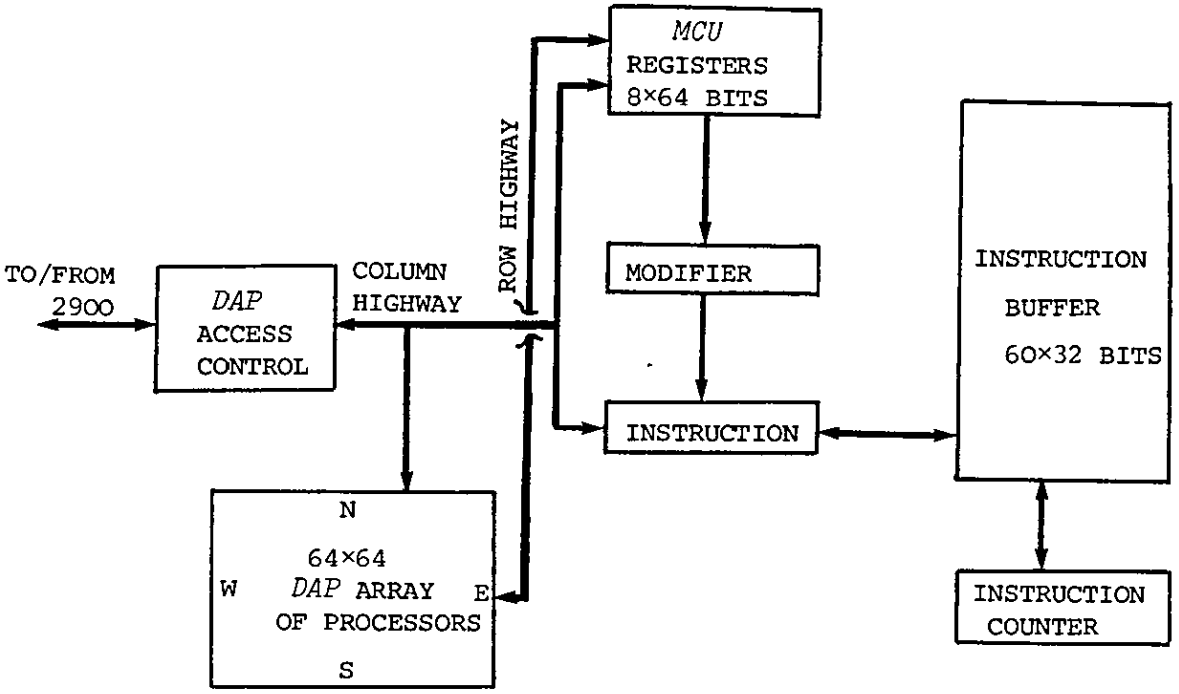


Figure I.B.3.2.3-f2: The Major Hardware Units of the *DAP* System.

I.B.3.2.3.i: THE ORTHOGONAL COMPUTER CONCEPT

Another major milestone in the history of parallelism is, undoubtedly, Shooman's concept of parallel computing using *vertical* data [*SHO060*]. Although we have already referred to some architectures which utilized this concept, we discuss this subclassification here to be precise with the *SIMD* genealogical topology.

Shooman, in 1960, proposed a new way in parallel processing achieved by referencing the memory not only in a word-serial/bit-parallel fashion, i.e. *horizontally*, but also in the orthogonal or *vertical* direction, i.e. across the words by bit slices; this effect could simplify many problems where the retrieval search was ended after searching only a few bits of each word. According to his notion, each processing element of the organization was assigned to each word occupying an horizontal row of the two-dimensional bit-space in the memory, thus making feasible the parallel

processing of the bits of a bit slice. This procedure formed the so-called bit-serial/word-parallel processing.

Of course, in addition to the 'Vertical Arithmetic Unit' (*VAU*), the orthogonal computer was provided with a 'Horizontal Arithmetic Unit' (*HAU*), to perform the word-serial/bit-parallel operation for the cases that would be more efficient.

In conclusion, the main representatives of the orthogonal concept, the *STARAN* and the *ICL DAP* computers, have been presented together in proposals for the recent construction of a 'Massively Parallel Processor' (*MPP*) with an (*128×128*) array of one-bit *PEs*, connected two-dimensionally.

I.B.4: THE GENEALOGY OF THE PIPELINED VECTOR† ORGANIZATION

The *pipeline* notion, in its widest sense, can be included in the general *parallelism* notion; in other words, it is a form or technique to imbed parallelism or concurrency into a computer architecture.

Pipelined vector computer architecture has received considerable attention since the 1960s when the need for faster and more cost-effective systems became critical. The merits of a Pipelined computer are its *availability* and *reliability*; also pipelining can help to match the speeds of various subsystems without duplicating the cost of the entire system involved.

The pipeline is closely related to an industrial assembly line. Like the assembly line, precedence is automatically honored, but, it takes time to fill the pipeline before full efficiency per cycle is reached and also time to *drain* the pipeline totally.

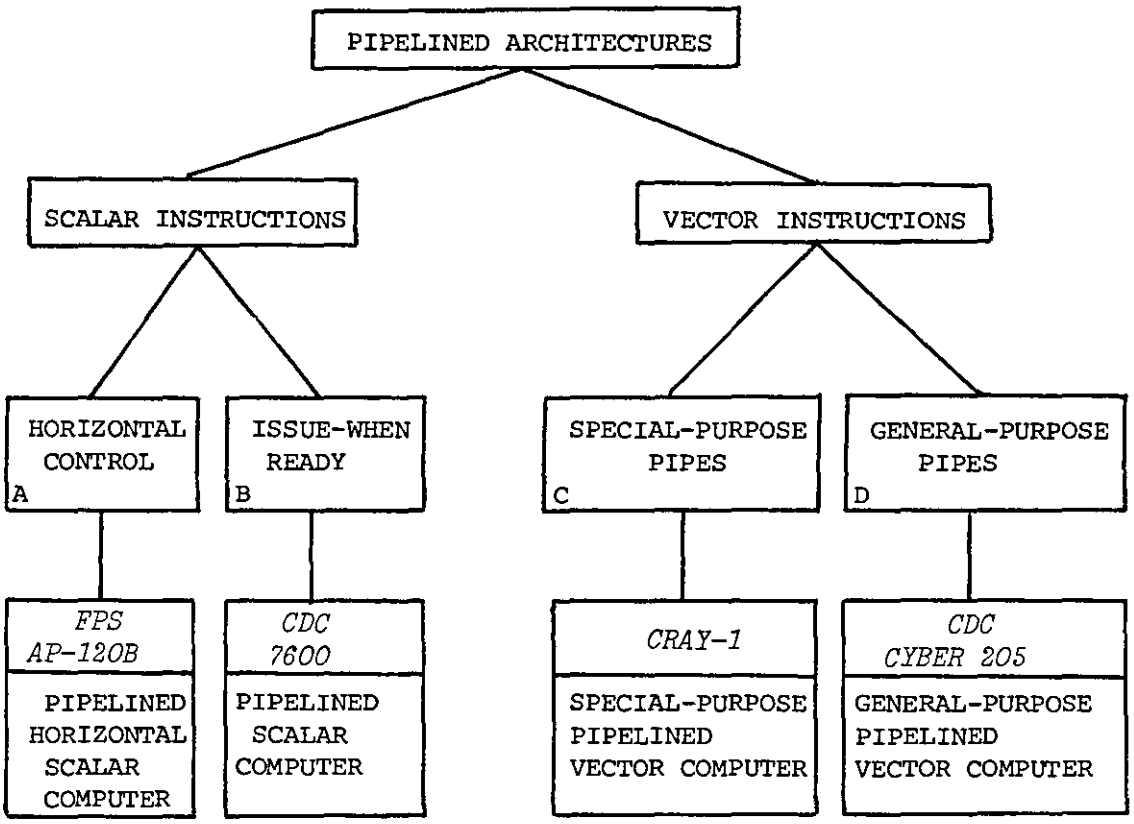
In *Figure (I.B.4-f1)* - (see Hockney [*HOCK81*]), in 1981, we can see the topology of the *genealogical tree* of the Pipelined architectures. Although Pipelined architectures are somewhat different compared to *SIMD* and Multiprocessor architectures, they present a significant interest because the algorithms best suited for *SIMD* systems are closely connected to those which achieve a great efficiency on a Pipelined system.

The future of pipelining can be considered most promising if we take into account the electronic and Large Scale Integrated (*LSI*) circuits evolution which offers faster and much cheaper hardware components.

In the following paragraphs, we shall present the theoretical considerations behind the pipeline notion, briefly surveying some commercial representative systems and establishing a *criterion* for the trade-off between sequential and pipelined vector processing.

---

†*Implies emphatically both, Pipelined 'Scalar' and 'Vector' architectures, appropriately distinguished when it is necessary.*



A: One instruction controls all units at each cycle (Horizontal control);

B: Instructions are issued to units individually, when they are ready to carry out an operation;

C: Separate special-purpose pipelines for each type of arithmetic operation;

D: One or more general-purpose pipelines, each capable of performing more than one type of operation.

Figure I.B.4-f1: The Genealogy of Pipelined Architectures.

I.B.4.1: Pipeline As A Fundamental Design Principle And Performance Characteristics

In this paragraph we present the principles of *overlap* and *pipelining* as general techniques for handling even precedence dependent tasks, operating at several levels of machine design.

Overlap is the phenomenon of concurrent processing, often towards some well-defined common goal. In a computer system there can be an overlap of input/output operations with processor operations, namely an *Asynchronous input/output*. Within the processor, there can take place an overlap on what the user realizes as a monolithic computer instruction; to the computer designer this is made up of a number of distinct operations, such as, fetching the instruction, decoding the operations involved and fetching the operands before it is finally executed. These operations fall into two separate phases or cycles of operation, the Interpretation or Instruction cycle (I), when an instruction is acquired and stabilized in the processor and the Execution cycle (E), when the particular function is actually performed.

More detailed overlaps can be designed into the machine; a simple example is the simultaneous running of many input/output devices. In the case of I and E overlaps an interlock mechanism is required between the different sub-units, each entrusted to carry out the smaller subprocesses, in order to enforce the precedence rules. This is achieved by the propagation of signals between the autonomous units indicating the completion of a task and the validation of registers at the interfaces. Although the interlocks can present a significant complexity, multiple processor overlap designs can potentially raise the performance of the computer several-fold.

Such an extreme form of multiple overlap is the *pipelining* technique, when rather than completion signals there are synchronizing time clock pulses reminiscent of *SIMD* parallel processing.

Pipelined or Vector computers achieve an increase in computation speed by decomposing or segmenting successive computational processes, each into several subprocesses, which can be executed efficiently by special autonomous and concurrently operating hardware units, which overlap their operations to

give an increased rate of completion of the process. This is possible even if subprocesses show a precedence dependence.

This technique, was first introduced in computers such as *Atlas* and *Stretch*. Although, very often, we read in publications the term *pipeline* in general, this concept depending on the degree of subdivision of processes can be implemented at several levels. Therefore, the subprocesses can be steps of the instruction execution cycle which means several partially completed instructions can be in progress simultaneously; in this case, although the time to complete any one instruction is still limited by the sum of the times for the various activities, the rate at which instructions progress through the *pipe* is only restricted by the time for an individual activity.

In the case of pipelining an arithmetic function, a subprocess can be one (or more) steps of the algorithm. Finally, the pipelining of software processes can be considered, thus extending the concept of a co-routine to that of *parallel co-routines*.

Particularly, Händler [*HAND82*], in 1982, distinguished three different logical pipeline levels, under the names: *macro-pipelining* for the program level, *instruction pipelining* for the instruction level and *arithmetic pipelining* for the word level. Furthermore, others distinguished the instruction pipelining, depending on the control structure of the system, to *strict* and *relax* pipelining; under the former notion they consider a pipeline with a smooth ordered data flow through it, whilst under the *relax* term they consider a pipeline which may accept a *turbulence* within it, e.g. later operations can move ahead of earlier operations. With the latter control structure the system is fully asynchronous, more powerful and flexible, but also more expensive.

In addition to the hierarchical levels of pipelining, the pipe itself can be further distinguished by its design configurations and control strategies into two forms, the *static* and the *dynamic* pipe. Sometimes a pipelined unit is dedicated to a single algorithm or function, e.g. a pipelined adder or multiplier; in this case it can be termed as a *uni-functional* pipe with a *static* configuration.

On the other hand, a pipelined unit can be dedicated to a set of different functions, namely a *multifunctional* pipe; in this case the pipe may have to be *flushed* between two consecutive and different operations. This sort of pipe can be either *static* or *dynamic*. The difference between them is that in the *static* case, at any time instant, only one configuration (i.e. interconnections of pipeline modules) is active and therefore any overlapping of operations has to involve the same interconnection; in *dynamic* multifunctional pipes several or all configurations can be active thus permitting a synchronous overlapping on different interconnections. The easier control of multifunctional static pipes justifies the fact that most, if not all, of the existing Pipelined computers utilize these kind of pipes.

The simplified model of a general Pipelined computer is depicted in *Figure (I.B.4.1-f1)*, comprising m pipelining segments; each of these segments performs its part of the processing and consequently the final result is obtained at the end of the last segment.

The *pipeline concurrency* will be exemplified considering the process of executing instructions. In *Figure (I.B.4.1-f2)* we considered four modules which individually can execute a subprocess into which the task of processing an instruction has been decomposed. In consequence four successive independent instructions may be executed concurrently. The overlapping procedure amongst the individual modules is depicted in *Figure (I.B.4.1-f3)*

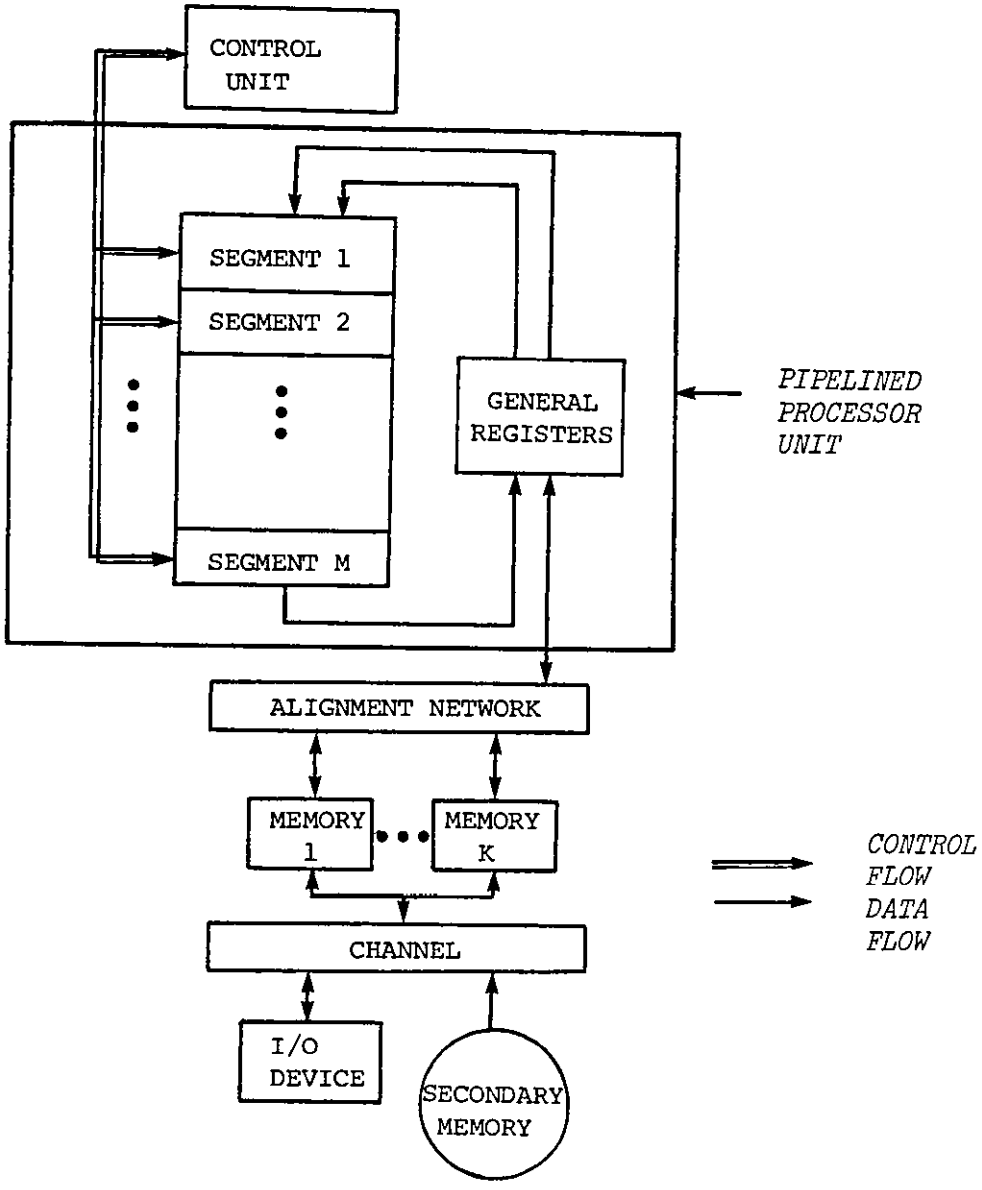


Figure I.B.4.1-f1: A Pipelined Processor System.

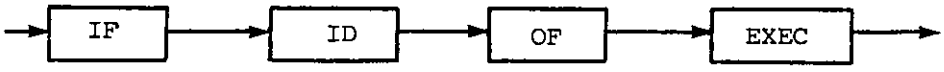


Figure I.B.4.1-f2: The Modules of a Pipelined Processor.

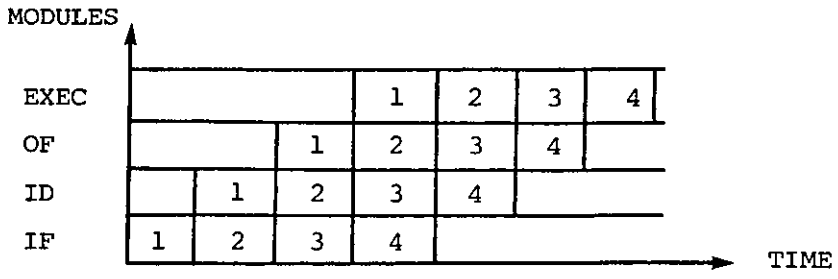


Figure I.B.4.1-f3: The Module-Time Diagram.

by a space (module)-time diagram. According to this, while the *EXEC* module is executing the first instruction, the Operand Fetch (*OF*) module fetches the required operand for the second instruction, the Instruction Decode (*ID*) module prepares the various activities for the third instruction and the Instruction Fetch (*IF*) module fetches the next instruction.

However, there exist some design and operational problems associated with a typical pipeline which can actually influence the efficiency and performance of the resulting design. These are the *buffering*, *busing structure*, *branching* and *interrupt handling*. Finally we shall briefly refer to the processing of arithmetic functions.

The use of *buffers* serves the purpose of continuing the smooth data flow through the pipeline segments whenever a variable speed occurs. In other words *buffering* or *look-ahead* is the process of storing results of a segment temporarily before advancing them to the next segment. Similarly to an industrial assembly line, when the slowing down segment resumes normal service, it clears out its buffer, perhaps, at a faster speed. Consequently, buffering may be required before or after any segment with processing speed not fixed, but anyway the expected full speed-up is not always achieved, since the buffers have to be stabilized before the activity transfer can be effected.

Excepting the architectural features of a Pipelined processor, the *busing structure* is a very important and decisive factor for the efficiency of an algorithm to be executed on such a system. Pipelining, theoretically, refers to the concurrent processing of *independent* tasks (e.g. instructions), which may be in different stages of execution due to overlapping. In real time, often, Pipelined computers have to tackle dependent or intermixed tasks. With dependent tasks, their input and traversal through the pipe have to be paused until the dependency is resolved.

The internal *busing structure* serves this purpose by routing the results to the requesting modules efficiently, thus reducing the adverse effect of tasks dependency. In the case of intermixed tasks, more concurrent processing can take place, since the resolving of dependency is hidden behind the processing of the independent tasks.

Another quite damaging factor, even more than the task (e.g. instruction) dependency, symptom which influences the performance of a Pipelined computer, is *branching*. The encounter of a conditional branch not only delays further execution, but also affects the performance of the entire pipe, since no one can tell the exact sequence of instructions to be followed until the deciding result becomes available at the output. To eliminate the damaging effects of branching, various different techniques have been employed to provide mechanisms through which processing can be resumed safely, since an incorrect instruction branching may create a discontinuous supply of instructions.

A similar effect, to the conditional branching, is caused by *interrupts*, namely a disruption of the instruction stream continuity takes place; the interrupt must be serviced before any action can be applied to the next instruction. In the case that the cost of recovery is not overly substantial, then sufficient information is set aside for the recovery of the next instruction; otherwise these two instructions have to be executed sequentially, a fact which radically changes the aim of pipelining.

In the case of the *STAR-100* processor, a recovery mechanism in the form of special interrupt counters is present; these are capable of holding addresses, delimiters, field lengths, etc., information necessary for the eventual recovery of vector-type instructions after an unpredictable interrupt has occurred. In a more general-purpose Pipelined computer the instruction recovery, after an interrupt, is a very costly and complex

problem. Also, different types of interrupts, depending on what they are associated with, can be distinguished, e.g. in the *IBM 360/91*, two types of interrupts occur, the *precise* interrupts, associated with an instruction (like an illegal operation code) and the *imprecise* interrupts resulting from the storage, address and execution functions.

Finally, the execution of *arithmetic functions* has been one of the most fruitful applications of overlapped processing in order to improve the total throughput. Specifically the advantages of pipelining will be obtained when floating point operations are being executed since they are quite long; but again the full speed-up will not be obtained unless all the modules of the pipe are fully used. For example, the pipe of the *TI ASC* system, the architecture of which is depicted in *Figure (I.B.4.1-f4)*, has eight modules to execute floating point instructions; a floating point addition on this system does not use modules 6 or 7 and therefore a reduced speed-up is obtained.

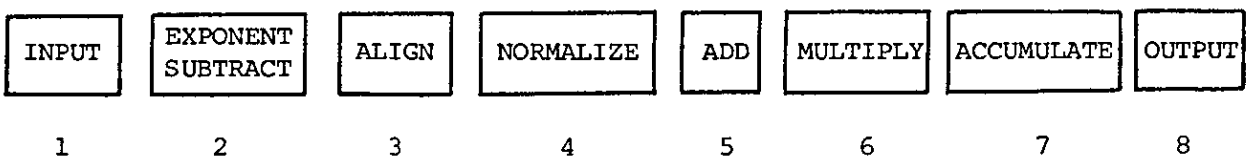


Figure I.B.4.1-f4: Modules for Floating Point Operations.

In conclusion, we present some performance considerations of Pipelined computers; at first, the evaluation of the basic timings related to the throughput rate, since this is one of the most important performance measures of such a system.

Let us suppose that, in an ideal case, each subprocess of the original process can be executed on a dedicated module or segment of the pipe in time τ, which is the same for all modules. If the pipe comprises p modules, then the process (subdivided into at most p subprocesses) will be executed

in $p.\tau$ time. In the case of k consecutive processes and due to the overlapping processing, the first process produces a result in $p.\tau$ time, whilst the $(k-1)$ subsequent results are obtained in $(k-1).\tau$ time. In other words, the k processes will be executed in $p.\tau+(k-1).\tau$ time, delivering a result at every interval τ, after a set-up time of $(p-1).\tau$. If we let t be the time to complete a single process in a sequential computer, then to achieve a faster execution of the k processes we require:

$$(k-1).\tau+p.\tau<k.t$$

$$\Leftrightarrow \qquad k > \frac{(p.\tau-\tau)}{(t-\tau)} \quad . \qquad (I.B.4.1:1)$$

This last formula expressed the condition which must be satisfied for a Pipelined system to be efficiently utilized; namely the number of processes has to be long relatively to the number of modules in the pipe. In real applications the throughput of a pipeline is determined by its slowest facility, or *bottleneck*. In this case the throughput can be improved either by subdividing the bottleneck element, or by placing facilities in parallel - see *Figures (I.B.4.1-f5,6,7)*.

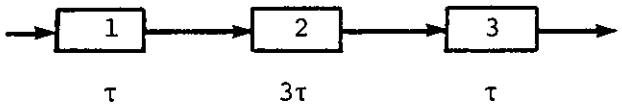


Figure I.B.4.1-f5: The Bottleneck is in Module 2.

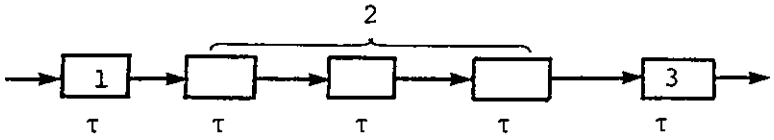


Figure I.B.4.1-f6: Subdivision of Bottleneck.

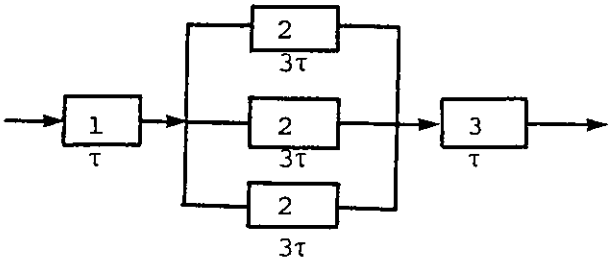


Figure I.B.4.1-f7: Paralleling of Bottleneck.

To formulate the *Speed-up* and *Efficiency* aspects of pipelining in real time, let us consider again k tasks which can be presented to the pipe without incurring new *set-up* and *flush* times. Let τ_i be the required processing time of a subtask, by the i^{th} module and τ_j be the bottleneck time, i.e. $\tau_i \leq \tau_j$, for $1 \leq i \leq p$, where p is the number of modules in the pipe. The *Speed-up* is given by the quotient:

$$\frac{T_s}{T_{pipe}} = \frac{k \cdot \sum_{i=1}^{p} \tau_i}{\sum_{i=1}^{p} \tau_i + (k-1)\tau_j} \quad . \qquad (I.B.4.1:2)$$

In ideal circumstances, when τ_i is the same for all modules, then formula *(I.B.4.1:2)* becomes:

$$\frac{T_s}{T_{pipe}} = \frac{p.k}{p+k-1} \quad , \qquad (I.B.4.1:3)$$

which at the limit ($k \to \infty$) produces a p-fold Speed-up, namely it approaches the pipeline length as would be expected. The *Efficiency* of the pipeline is defined by the quotient:

$$E_{pipe} = \frac{\textit{Time spent on computing}}{\textit{Time modules are used}} = \frac{k \cdot \sum_{i=1}^{p} \tau_i}{\sum_{i=1}^{p} [\sum_{i=1}^{p} \tau_i + (k-1)\tau_j]} \quad , \qquad (I.B.4.1:4)$$

which in ideal circumstances becomes:

$$E_{pipe} = \frac{k}{p+k-1} \quad . \qquad (I.B.4.1:5)$$

From the last formula it can be easily concluded that for $k >> p$, the Efficiency tends close to the theoretical value *one*.

The mathematical explanation of the Speed-up and Efficiency formulae is presented in *Appendix C-I*.

I.B.4.2: VECTOR PROCESSING CHARACTERISTICS

The most natural application for pipelining is *Vector* processing; the main requirement in justifying the pipelining of a process is the frequent repetition of the same sequence of operations which occurs naturally in vector processes. The overlapped characteristics of pipelining are employed when the required transformation of vector elements are independent to each other.

A *Vector pipe* can be characterized by the existence of one or more, static or dynamic, multifunctional pipes in the execution unit (arithmetic and logic unit). Natural overheads are associated with the vector processing, such as the *set-up* time, or the time to structure the pipeline preparing the vector operand streams and the *flush* time which is the period of time between the initial operation (the decoding) of the instruction and the exit of the result (for vectors, the first result element) through the entire pipe.

Although the way to treat vector operations is different from system to system, basically all vector instructions must produce the information of the type of the performing instructions, (e.g. floating-point add), as well as, the start address of each vector and the increment of this address to find the successive elements to be processed. The alternatives appear when some of the vector elements are to be chosen according to some control vector, or when the routing of a resulting vector is required due to the way it is going to be used as the input operand for the next instruction.

According to the above time constraints, the vector processing time, in the case of an effective vector field length k, is given by:

$$T_p^v = T_{s/p}^v + (k-1).\tau_j^v + T_{f/h}^v, \quad (\tau_j^v - \text{bottleneck time}). \qquad (I.B.4.2:1)$$

On the other hand, the execution of k operations in the same pipe, but in

scalar mode, i.e. without vector processing power, is given by:

$$T^{s} = (k-1).\tau_{J}^{s} + T_{f/h}^{s} \quad . \qquad (I.B.4.2:2)$$

The *bottleneck* time τ_{J}^{s} will be greater than τ_{J}^{v}, since the additional register settings for the vector instructions are very advantageous and they will speed-up the operand fetches, as well as the computations of their effective addresses. From the last two formulae we deduce the advantageous case for vector processing:

$$T_{s/p}^{v} + (k-1).\tau_{J}^{v} + T_{f/h}^{v} \leq (k-1).\tau_{J}^{s} + T_{f/h}^{s}$$

$$\Leftrightarrow \quad T_{s/p}^{v} + T_{f/h}^{v} - T_{f/h}^{s} \leq (k-1).(\tau_{J}^{s}-\tau_{J}^{v}) \quad . \qquad (I.B.4.2:3)$$

The last formula *(I.B.4.2:3)* reveals that *Vector* processing is beneficial when the length of the executed vector is considerably large; in other words, if the *set-up* and differential *flush* times are large compared to the differential bottleneck times, then a large vector field length is required to justify vector processing.

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

In conclusion, the advantages of *Vector* processing as compared to a *sequential* Pipelined Processor (e.g. *I.B.M. 360/91*) are its speed improvement for considerably lengthy vectors and the more efficient utilization of all the system resources when dealing with vectors, which is a result of the more orderly management. The overhead incurred is principally in the additional software facilities required to use the pipeline efficiently. There is also additional control circuitry required, especially for Vector computers with multifunctional pipes, to establish the desired configuration and the routing of operands between pipe segments.

When these costly problems have been solved efficiently,then vector processing would be generalized and extended to smaller scale processing systems.

I.B.4.3: Implemented Pipelined Vector Computers

In this concluding paragraph of the Pipelined processors, we briefly refer to the architectural characteristics and performance of some commercially implemented Pipelined systems.

The *CRAY-1* was one of the most successful Vector computers with a design philosophy following closely the tradition of the *CDC 6600* (renamed *CYBER 70* model 74) and *CDC 7600* (renamed *CYBER 70* model 76). The *CRAY-1* is manufactured[†] by Cray Research Inc., at Chippewa Falls, Wisconsin, U.S.A., and its most striking feature is the small size. It comprises twelve independent pipelined functional units to perform the arithmetic and logic; these incorporate vector processing capabilities and can be *connected* to form efficient chains as a continuous pipeline, thereby maximizing overlapped vector processing. The maximum size of the main memory is 2^{20}, 64-bit words of bipolar memory with a 50ns access and cycle time, divided into 16 memory banks being capable of simultaneous operation. The memory has been increased to 4M words on the *CRAY-1S*[‡], divided into 8 or 16 memory banks. The maximum computing rate on the *CRAY-1* is 160M flops/s (i.e. 80 million multiplications and 80 million additions per second), with a clock period of 12.5ns. The *CRAY-1* was designed without any incentive for experiments with new technology; the aim was the commercial competitive substitute of the existing computers, such as the *CDC 7600* and the *IBM 360/195*.

The *CYBER 205* is the culmination of a long programme of research and development that initiated with the design and delivery of the *CDC STAR 100* computer in the period 1965-75. It is manufactured by Control Data

[†] *The first delivery was made to the Los Alamos Scientific Laboratory, New Mexico, in February 1976.*

[‡] *This series of computers was announced in 1979.*

Corporation in Saint Paul, Minnesota, U.S.A. and the first customer was the U.K. Meteorological Office, at Bracknell, in 1981.

The *CDC STAR 100* had many disadvantages which made it unattractive to potential customers. *CDC* decided to develop a new *LSI* technology and re-engineer the whole system using it, but retaining the software developed for the *STAR 100*. At first they manufactured the *CYBER 203*, initially described as the *STAR 100A*, which overcame the disadvantages of slow main memory and four times slower scalar arithmetic compared to the *CDC 7600* and *IBM 360/195*. The next step produced the (initially known) *STAR 100C* or *CYBER 203E* which was finally announced as *CYBER 205*. The clock period is 20ns and the number of pipes may be optionally increased to four (instead of two), the memory to 4M words and the I/O channels to 16, with an effective maximum performance of 800M flops/s in 32-bit arithmetic on a four pipe machine. However, the disadvantage of *unit vector increment*, caused by the *contiguous* vector requirement, i.e. successive vector elements should be stored in successive memory locations, remained.

The *AMDAHL 470V/6*, manufactured by AMDAHL Corporation, was the first computer to use *LSI* technology for its logic, utilizing a comparatively simple instruction pipelining architecture served by four functional units. The first *AMDAHL 470V/6* delivered in 1975 with a basic cycle time of 32.5ns, which was reduced to 29ns in the subsequent version of the *AMDAHL 470V/7*. The former model comprising four execution subunits (i.e. multiplier, adder, shifter and byte mover) performs 4.6M flops/s and the latter 7M flops/s, namely 1.2 to 1.4 times faster than *IBM's 3033*. Also a high-speed buffer (or *cache*) bipolar memory of 16K bytes (65ns access) was used to improve the effective access to the slow main memory of up to 8M bytes of MOS store (650ns access).

The *TI ASC* system, manufactured by the Texas Instruments, is closer to

the *STAR 100* than to *CRAY-1* and started around 1966 as a computer suitable for the high-speed processing of seismic data; it comprises four pipes, each capable of performing all the elementary instructions on vector operands. With the four pipes operating optimally a design rate of 50M flops/ /s was theoretically achieved. The semiconductor memory has 8 banks and a cycle time of 320ns.

In conclusion, the *FPS AP-120B*, manufactured by Floating Point Systems Inc., in Beaverton near Portland, Oregon, U.S.A., in 1976, is called an 'Array Processor'(*AP*)[†], since it is designed to process efficiently arrays of numbers. One might say that the *AP-120B* is to a *mini* or *medium* computer what the *CRAY-1* is to a large mainframe computer; in other words, a *poor-man's CRAY-1* due to its low cost. The machine is driven synchronously from a single clock with a period of 167ns. The standard memory has an access/ cycle time of 500ns, whereas the optional fast memory has a cycle time of 333ns. The system performs 38-bit floating-point arithmetic in separate pipelined multiplication and addition units and 16-bit counting and address calculation in an independent integer arithmetic unit. *Three* memories (for data, tables and programs) and two *scratch pads* of registers are provided, with multiple paths between each memory and each arithmetic unit. Typically, processing rates of 5-10M flops/s may be achieved.

An improved version of the *AP-120B* was announced in 1980 under the name *FPS-164*[‡]; the principal improvements compared to its predecessor 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 12M bytes.

[†] *This name does not imply that the computer is architecturally an array of processors.*

[‡] *Before FPS-164, an enhanced version of AP-120B, with more memory, called AP-190L, was designed.*

Finally, the arrangement of control in *AP-120B* computer was referred to as *horizontal microcode*, since each instruction, 64 bits wide, controlled the operation of all units in the machine every clock period; in other words, there was only one instruction in the instruction set with fields controlling each of the functions, although some fields overlapped excluding certain combinations of functions.

I.B.5: THE MIMD MULTIPROCESSING ARCHITECTURES

For a number of years Multiprocessing systems were relatively a rarity found primarily in special-purpose systems requiring high availability, e.g. in military command and control applications. The definition of a Multiprocessor as it appears in an Information Processing vocabulary says: *A computer employing two or more processing units under integrated control.* However, this definition is hardly complete since it does not refer to the *sharing* and *interaction* of mostly significant features consisting of the core of a Multiprocessing system.

Enslow [*ENSL77*], in 1977, provided a complete definition of a *true* Multiprocessor depending on its characteristics; according to him such a system comprises two or more processors with approximately comparable capabilities, all of them *sharing* access to a common memory, I/O channels, control units and devices. Also, the entire complex is under the control of a single operating system providing the *interaction* characteristic amongst the processors and their programs at the job, task, step, data set and data element levels.

In general, Multiprocessor systems consist of a subclass of *MIMD*- 'Multiple Instruction stream, Multiple Data stream', multiple-computer architectures. The *MIMD* computers since they possess a greater inherent flexibility than *SIMD* computers, are suitable for a much larger class of computations; these can be fitted in a much more straightforward way than for *SIMD* computers, but a careful synchronization of the assigned computations to the processors is required, for a high-efficiency to be obtained. This is a sharp difference between the *MIMD* and *SIMD* computers, where, in the latter ones, the synchronization is done automatically and additionally the task allocation problem of the former systems does not exist since all of the processors perform the same task.

Consequently, the trade-off between these two classes of computers lies on the problems which do not arise in *SIMD* computers due to the imposed constraints; such problems are eliminated in *MIMD* computers at the cost of the added flexibility, which comes with the need to solve the synchronization and allocation problems.

In the subsequent paragraphs, we refer to the reasons which urged scientists towards the design of this sort of architectures, focusing mostly on some significant hardware and software features of these systems that are necessary to support parallelism.

I.B.5.1: THE MIMD HARDWARE SYSTEM ORGANIZATION

A *MIMD* system can be defined as a collection of minicomputers (i.e. a *multimini*) or microprocessors (i.e. a *multimicro*), which are connected either through a *shared* memory or via low- or high-speed *data links*.

Although the actual implementation of this definition appears to be primarily a hardware design problem, this appearance is very deceptive; this is because implementing such an idea, aiming to support parallelism, suitable operating systems are required, as well as ultimate applications to make the complex an effective parallel computer.

Each component (or processor) of the *MIMD* complex provides its own control unit, thus being capable of generating its own stream of instructions, which can then be executed on its own stream of data, concurrently. Consequently, a *MIMD* complex is considered as an *asynchronous* system. The general structure of a *MIMD* complex comprising p-processors is depicted in *Figure (I.B.5.1-f1)*.

The shared memory may be a *multiported* main memory, *cache* memory, or a *multiported* disk. A bit-serial or parallel bus data path can be used to connect the I/O ports of two processors, or a shared bus data path in the

case of two or more processors; sometimes the broadcasted data, onto the bus, are intercepted by the 'interested' component, whilst in some other communication links (e.g. *daisy chain*) the information may pass through from one leg of the bus to the other, until the 'interested' component finally 'erases' the message off the path.

The architectures employing the shared memory interconnection have been coined *directly* or *tightly coupled*; this memory can be regarded as very fast, comprising several parallel memory units. More comprehensively, in such a system, all the processors can have access to all the memories, sharing I/O and other resources of the complex (i.e. peripherals), whereas the interprocessor communication latency is low due to the potential access time being limited only by the actual memory access time.

On the other hand, *indirectly* or *loosely coupled* systems differ from *tightly coupled* systems in that they do not share a common primary memory; they have disjoint, primary or main memory address spaces,which implies the existence (at the hardware level) of an explicit interprocessor communications interface, which in turn causes a higher latency of communications amongst the processors than would be caused when sharing a primary memory. Additionally, the execution of processes on *loosely coupled* systems can be performed asynchronously, whereas the most integrated *tightly coupled* systems require synchronization amongst cooperating processes. In *Figures (I.B.5.1-f2,f3)* the *loosely* and *tightly coupled* systems are illustrated. According to the above, *SIMD* architectures can be considered as *tightly coupled* systems, whilst most of the *MIMD* architectures as *loosely coupled* systems.

Although there is a control unit connected with each processor, a higher level control is required to take care of the data transfer and the task and various sequences of operations assignment amongst the processors.

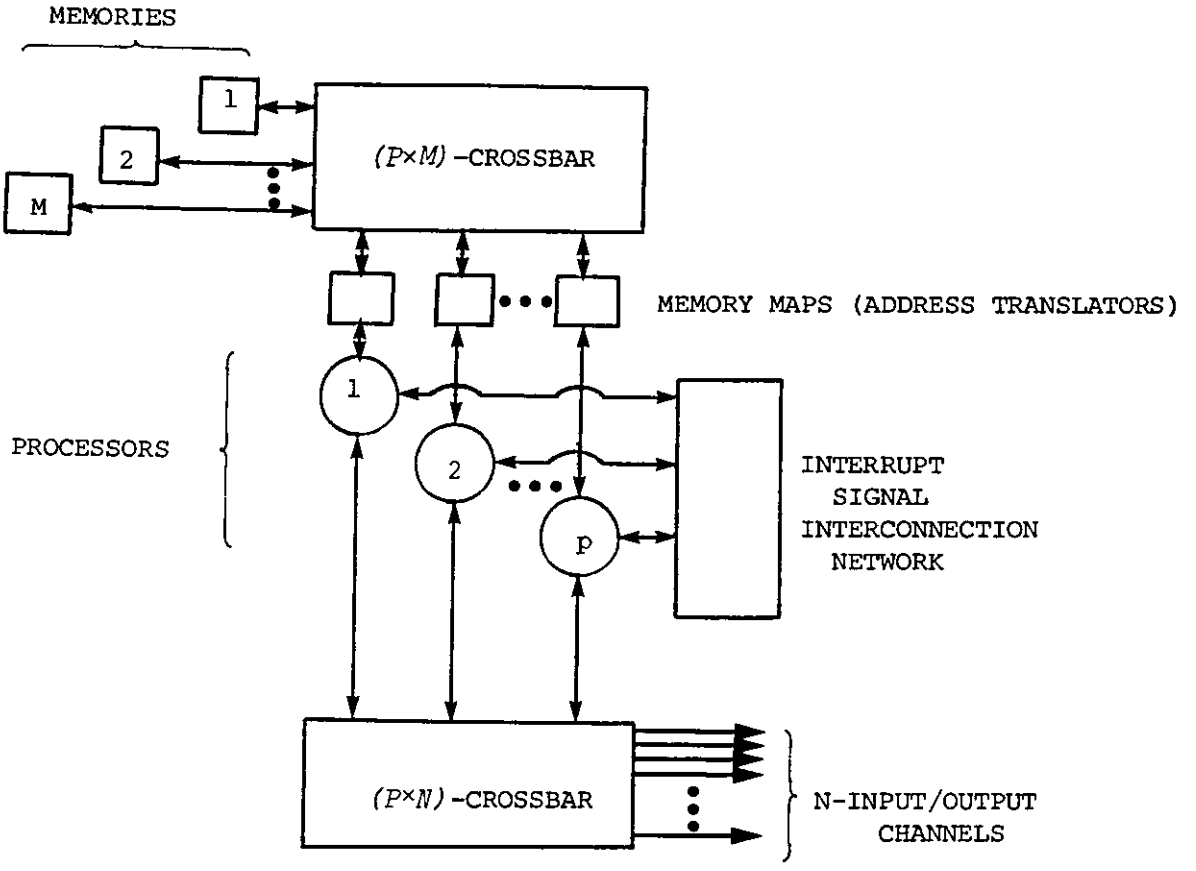


Figure I.B.5.1-f1: A *MIMD* Architecture.

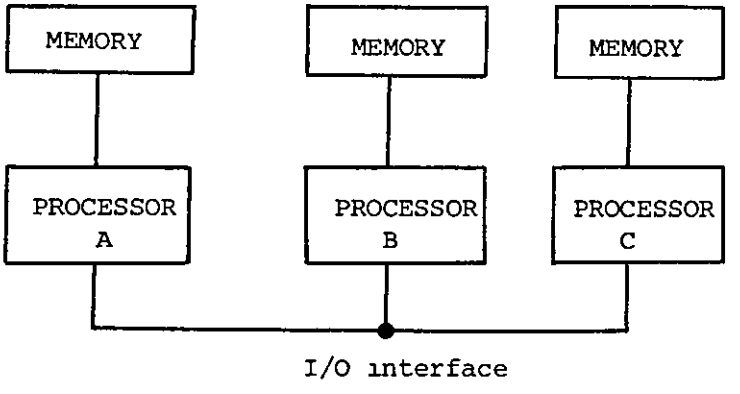


Figure I.B.5.1-f2: Indirectly or *Loosely Coupled* Systems.

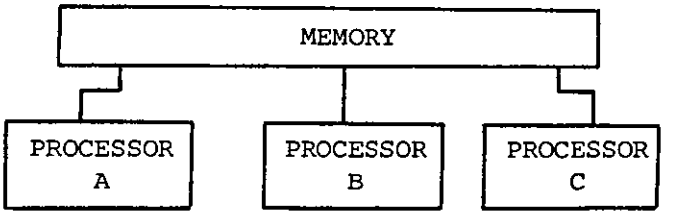


Figure I.B.5.1-f3: Directly or *Tightly Coupled* Systems.

This sort of control function can be provided by one of the processors, designated as the *global* processor, being given overall responsibility and connected via an I/O interface to every other *local* processor[†]. The global processor is the *entry* for all jobs into the system, but anyway it is not irreplacable, since in the case of a *crash* the overall control can be reassumed by one of the local processors.

In the case that the interconnection amongst the processors (e.g. minis) is restricted to the transfer of data files only and additionally there is no *global* controlling processor in the system, then we talk about *fully distributed* systems.

In a Multiprocessor system, the total throughput depends critically on the degree of memory access interference (or *conflicts*), fact which implies that the speed-up factor will always be less than the number of processors, due to that interference.

The memory access conflicts can be distinguished into two types, the *hardware* and *software* type. Since only one access can be made (per memory cycle), the former type of conflicts occurs when in a single memory cycle more than one processor attempt to access the same memory unit (or module) concurrently; the other requests must wait usually for a cycle or two in each case. The more of this type of conflicts that occur, the greater the degradation of the system, which is known as 'interprocessor interference'.

The latter type of conflicts occur when a processor attempts to use data which is currently being accessed by another processor. This data set forms the so-called *critical section* (see *Chapter II*); there is a *lock* mechanism which is activated by the 'served' processor at that time, preventing any other processor from accessing the same data set. This type of conflicts

[†] *As in the NEPTUNE system.*

are often known as *memory lockouts* and when processors encounter them they have to 'wait' and repeatedly check the status of the lock until the *unlock* state is reached.

To reduce memory conflicts the use of a very fast buffer memory *(cache)*, interposed between the processor and the main memory, has been proposed. In this memory can be stored the very frequently used data and/or instructions. Specifically, the *cache* memory contains the information of a limited number of contiguous memory locations and the whole notion of it depends on a probabilistic theory for accessing this information. Certainly a single *cache* may be proved insufficient to support a Multiprocessor system, since it must match the speed of several processors. The number of utilized *cache* memories in an architecture certainly depends on the organization chosen. Consequently, systems can be configured either with a *cache* associated with each memory and accessible to each processor via a crossbar switch, or with a *cache* associated with each processor and capable of being loaded from any memory. Apparently in the latter configuration the contention is reduced since only one processor has access to the *cache* and references, code and data in it alone for most of the time.

A difficulty presented by *cache* memories is that of the corruption of shared data, due to the fact that at least two processors may alter independent copies of the same data, in different ways, in different *cache* memories, thus creating a contradictory confusion in the main store. In addition, since it can rarely be predicted which specific information will be used next, a better way to reduce memory conflicts is to utilize simple *private memories*, each associated with each processor, in which *local* independent copies of the information to be used are stored.

Certainly all these communication and conflict problems justify the fact that large numbers of processors cannot be utilized effectively in *MIMD*

systems; in fact, most of the *MIMD* systems comprise at most 16 processors†. Anyway the interconnection network for such a system, with a small number of processors, is quite manageable, although a *reduced* interconnection pattern can be used, similarly to *SIMD* systems, in cases when the complex comprises many processors.

In conclusion, shared memory I/O systems are characterized by the following general types of *physical* and *logical* interconnection schemes:

- Time-shared/Common bus
- Crossbar Switch
- Multibus/Multiport memory

} *Physical*

- Virtual
- Mailbox

} *Logical*

to be briefly discussed in subsequent paragraphs.

I.B.5.1.1: The Time-Shared/Common Bus Interconnection Schema

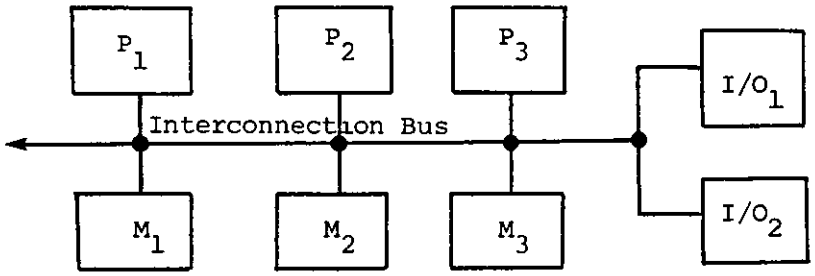
Figure (I.B.5.1.1-f1) illustrates this interconnection scheme which is the simplest one for Multiprocessors, since all processors, memories and I/O units are connected to a single bus which incorporates some arbitration logic, to serialize concurrent requests by a number of processors. Fixed priorities, i.e. 'First-In, First-Out' (*FIFO*) and 'daisy chaining' are used to resolve processors contention since the common bus is a shared resource.

The reliability of such a system and the interference amongst the processors requesting the bus, are its most serious limitations, together with the total overall transfer rate within the system, due to the limited bandwidth and speed of the single path. For this reason, private memory and private I/O are highly advantageous, although a common path failure implies a complete system failure. Processors are connected to the common bus via

†*The NEPTUNE system to be described later on comprises four processors.*

private buses, used to interconnect a central processor, some private memory, a number of I/O interfaces, as well as, a bus switch interface.

On the other hand, the *shared* environment of the common bus, except the memory arbitration logic, some shared memory, shared I/O and a bus switch interface, contains the *List Controller* or else *Concurrency Box* used to provide a synchronization capability by means of semaphores and queue operators; an example of such a system, utilizing multiple mini-computers, is the *SL-10* Data Switch for the Bell Canada Datapac network (see Weitzman [*WEIT80*]).



P : Processor Element
M : Memory Element
I/O : Input/Output Controller

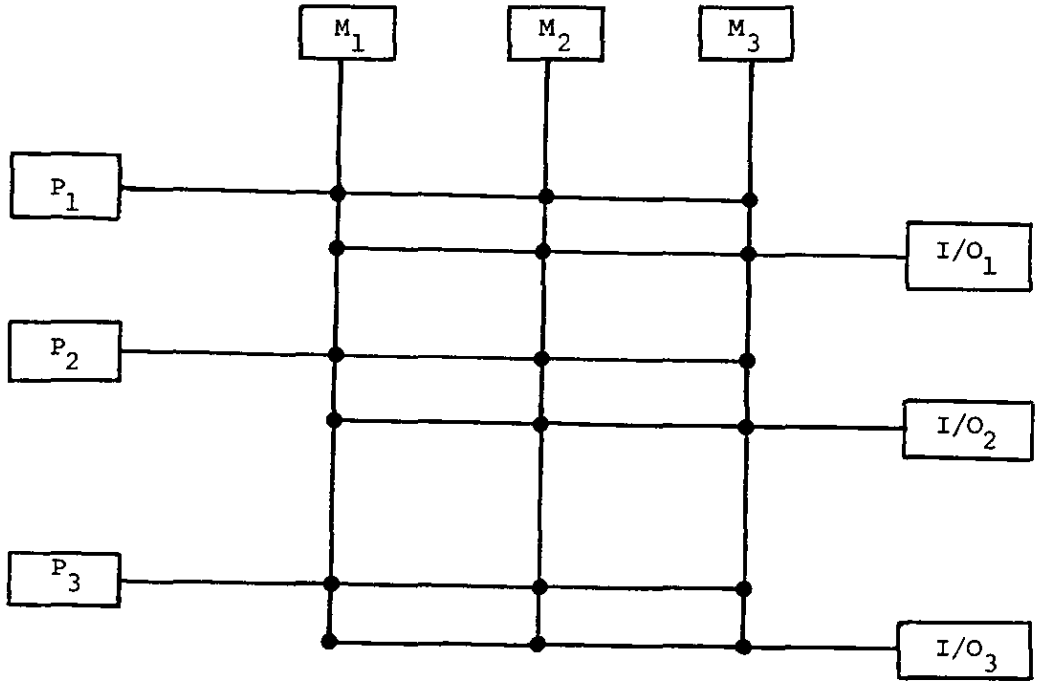
Figure I.B.5.1.1-f1: The Time-Shared/Common Bus Interconnection Schema.

I.B.5.1.2: THE CROSSBAR SWITCH MATRIX INTERCONNECTION SCHEMA

The *crossbar switch matrix* can be considered as the most extensive and expensive interconnection scheme providing direct paths from processors to memories. In the case that the system comprises p processors and m memory modules, then the crossbar requires $(p \times m)$ switches, each providing hardware capable of switching parallel transmissions and for resolving conflicting requests for a given memory module; this fact proves the switching device to be the dominant cost factor of the entire complex. This will become more obvious with the advances in *LSI* technology, which will minimize the memory and processor build cost more rapidly compared to that of the switch structure.

This interconnection scheme can be obtained from the previous organization by increasing the number of buses, so that individual paths are available for each memory and I/O unit. The *bus interface logic*, required by the functional units, is minimal, since they perform neither recognition of data intended for them, nor conflict resolution. These functions are performed by the switch matrix, which consequently is very complex (the complexity grows exponentially as p,m become large), costly to control and physically large. Every line in the crossbar scheme contains address, data and control bus signals which may correspond to as many as 32 to 64 wires.

However, the major merit of such a crossbar switch, which is depicted in *Figure (I.B.5.1.2-f1)*, is the concurrent transfer ability amongst all processors and memory modules.



P : Processor Element
M : Memory Element
I/O : Input/Output Controller

Figure I.B.5.1.2-f1: The Crossbar Switch Matrix Interconnection Schema.

For historical purposes, the earliest known system which employed a crossbar-type interconnection switch was the Ramo-Wooldridge *RW-400*, the *Polymorphic Computer* system, developed for the U.S. Air Force for large command and control installations.

I.B.5.1.3: The Multibus/Multiport Interconnection Schema

This organization is reminiscent of the crossbar switch, except that the control, switching and priority arbitration logic are concentrated at the interface to the memory modules. There are private buses for each processor through which it can access all memory modules. All memories and I/O units (*passive* elements) have *multiple ports*, one for each connection to a processor; the memory access conflicts, which are bound to occur, are resolved by assigning fixed priorities to each memory port.

This organization due to the high offered throughput capability is very often used, although the flexibility of the system is somewhat limited since the number of memory ports restrict the number of processors to which it can be linked.

The complexity and amount of hardware required is of the same order of magnitude, as in the crossbar switch, namely m connections per memory module, but still it is more localized. In *Figure (I.B.5.1.3-f1)* the outline of this organization is illustrated.

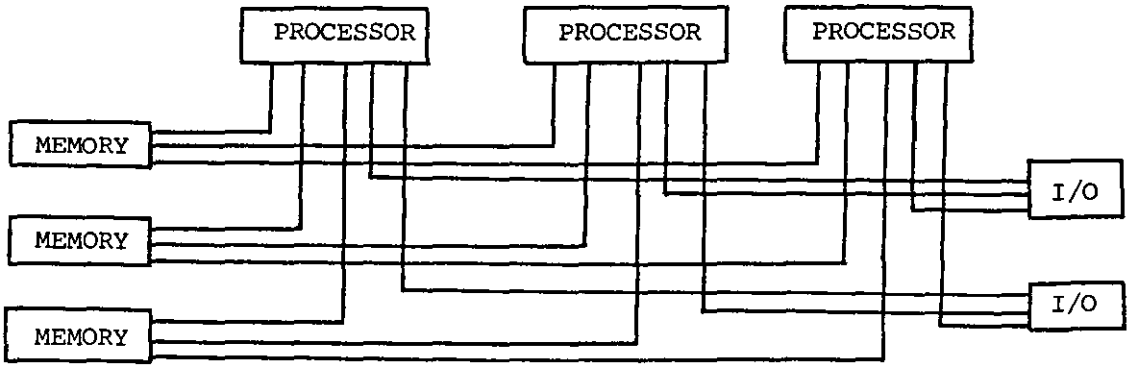


Figure I.B.5.1.3-f1: The Multibus/Multiport Interconnection Schema.

I.B.5.1.4: The Virtual And Mailbox Logical Interconnection Schemas

There are two major forms of shared memory logical interconnections, through which many of the multiport schemes, as well as other interconnection schemes, can also be represented; these forms are the *virtual* and the *mailbox* shared memory.

The former incorporates a shared memory into a virtual memory environment; according to this, there is a virtual address space, beyond the size of any processor's real memory, to provide addressing capability to a larger memory space, for the case that more active computing elements exist. Of course, there is an unavoidable overhead resulting from the requirement for address *translator* hardware, and the need for various segmentation techniques. An example of a virtual shared memory minicomputer system is *PLURIBUS* developed by Bolt Beranek and Newman (see Weitzman [*WEIT80*]), utilizing a number (typically from 6 to 14) of Lockheed *SUE* minicomputers to achieve its processing power.

In the latter form of logical interconnection, the shared memory acts as a *message* center for the intercommunication of various *CPU's*. The use of local memory is highly recommended, in an attempt to minimize the common memory reference, in the case that the number of processors exceeds three; this is due to severe contention problems which will arise when processors access the common memory for a substantial fraction of its cycles.

In conclusion, this scheme is more efficient than the former[†] one, in cases when the memory access characteristics, of the specific application, are well known, as in dedicated real-time systems; additionally, in this scheme there is no extra overhead as with the address translator hardware in the virtual shared memory.

[†] *A detailed comparison between all these interconnection schemas can be found in [WEIT80], pp.37-38.*

I.B.5.2: THE MIMD OPERATING SYSTEM ORGANIZATION

After the presentation of the basic *MIMD* hardware interconnection schemas and problems, in order to complete the picture of such a system, we briefly discuss the basic organizations in the design of Operating Systems, which are needed to control a Multiprocessor.

The difference between the software requirements of large systems utilizing multiprogramming and Multiprocessors, are conceptually very little; actually, from the most common functional capabilities required in an Operating System, such as, resource allocation and management, table and data set protection, prevention of system deadlock, abnormal termination, I/O load balancing, processor intercommunication, processor load balancing and reconfiguration, only the last three may be considered as unique to Multiprocessors. The efficiency of an Operating System is very significant in such a system, otherwise a poor performance could destroy any cost-performance advantages that the system has achieved.

In the Operating System the code segments that provide the fundamental services (e.g. interrupt decoding) to the system are often grouped under the collective name of the *kernel* or the *executive*.

There are three basic organizations in the design of Multiprocessors Operating Systems, 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 is the easiest one since it can be obtained straight from a uniprocessor Operating System with full multiprogramming capabilities, by making relatively simple extensions to it. This justifies the fact that, at first, most of the Multiprocessor systems utilized such an organization. However, this type is very inefficient in utilizing and controlling the system's resources; additionally, the

[†] *The NEPTUNE system, to be discussed later on, utilized such an organization, although modifications are in hand to produce an Operating System more likely to the symmetric organization.*

difficulty to construct a fast *master* to keep the *slave* busy created a delay time and consequently high performance was not very likely to be achieved.

In this organization it is not necessary for the executive and the routines it uses to be replicated, since only one processor will be using them, a fact which minimizes the table conflict and lock-out problem for control tables. If the *slave* requires a service to be provided by the *executive*, it requests that, waiting until an interruption of the current program on the *master* processor and the dispatch of the executive. However, in the case of an irrecoverable error or failure of the *master* processor this implies the catastrophic failure of the entire system, which can then be restarted only by the operator's intervention. On the other hand, this organization, which requires simple software and hardware, is most effective for special applications with work load well defined or for asymmetrical systems with *slaves* having less capability than the *master* processor.

In the case of the *separate executive* organization each processor serves itself due to the replication of some of the supervisory code to provide distinct copies for each processor. Consequently each processor (actually each *executive*) has its own set of I/O equipment (reconfigurable by manual intervention), files and private tables; though, there are some tables which must be common to the entire system thus creating table access control problems. Additionally, a failure case is not a complete system's failure although it can be a difficult task to restart the failed processor.

The philosophy of a *true* Multiprocessor system is more closely approached by the *symmetric* organization. In this organization all tasks in *kernel* software are treated *equally*; in other words, all processors are considered as an *anonymous* pool of resources, each of which can execute a *kernel* task as and when required. In this scheme the *master* passes from

one processor to the other, with an improved balanced load over all resources, whilst priorities set under static or dynamic control resolve the service request conflicts. Also, most of the supervisory code is reentrant since several processors can execute the same service routine simultaneously. Of course, the unavoidable conflicts, in multiprocessing, due to table accesses, as well as, the table lock-out delays, are present, but under control to protect the system's integrity. The advantages of this type of organization over the previous ones are, the better availability of a reduced-capacity system, true redundancy, the most efficient use of the available resources and a graceful degradation (i.e. the ability to reconfigure a viable system automatically from functioning components, after the failure of some others).

Finally, it must be emphasized that most Operating Systems for Multiprocessing architectures are not 'pure' examples of any one of the three schemes above; most of the implemented architectures have adopted 'hybrid' solutions combining features of all of them. The development of software systems for Multiprocessors is still in an experimental state.

I.B.5.3: IMPLEMENTED MIMD ARCHITECTURES

In this part of *Chapter I*, we shall briefly discuss the characteristics of some implemented *MIMD* architectures; certainly there are many such systems, built by various manufacturers (e.g. *Burroughs D-825*, *RCA 215*, *MIT/IL ACGN*, *Bell Labs. CLC*, *Plessy system 250*, etc.), but since this is not a detailed survey of the implemented *MIMD* systems, we shall restrict ourselves to some fundamental representative architectures such as the *C.mmp* and the Clusters of Microcomputers: *Cm**, both developed at Carnegie-Mellon University, U.S.A. In the subsequent paragraphs we emphasize on two other examples sited in Loughborough University, the *Interdata Dual processor*

and its successor, the *NEPTUNE* system, on which the majority of the research was carried out.

The *C.mmp* system (see Wulf, Bell [*WULF72*]), illustrated in *Figure (I.B.5.3.-f1)* was a major research project utilizing a crossbar interconnection system; the aim of this architecture was the investigation of economical techniques for interconnection, as well as the study in depth of the Operating System and the overall system's performance. The processors utilized in the architecture are various models of the *DEC PDP-11* and each of them has associated with it a block of dedicated private memory; also, each processor has associated with it a separate unit, the *address translator*, to translate addresses at the processor into physical memory addresses, for all accesses to the shared memory, since the address space of the main memory greatly exceeds that of the *PDP-11* itself. Finally, with each processor an I/O device is associated, which cannot be shared.

The shared primary memory comprises up to 16 modules $[M_0,\ldots,M_{15}]$, each consisting of 64K, 16 bit words, whilst the total size of the physical shared memory is 2^{20} words (or 2^{21} bytes). The use of a *cache* memory between the address translators and the crossbar switch can reduce the conflicts from accessing the memory, but if a large number (15-30) of processors is used, then the interference problem has a high cost-effectiveness anyway, since all the programs are stored in the common memory. The local memories are utilized to support the error-recovery and interrupt only, but not to store programs or data.

A disadvantage of the system is considered to be the high price of the required sophisticated crossbar switch, since the routing of the memory requests from the processors to the shared memory is done by the hardware within this switch; the very complicated hardware lowers the reliability of the system and reduces the performance speed because of queueing delays in the switch.

A more sophisticated shared memory, virtual environment, is the *Cm** system (see Swan, Fuller and Siewiorek [*SWAN77*]) developed at Carnegie-Mellon University. The basic 'building block' of this architecture is the 'computer module'. The main constituents of this module is a *DEC LSI-11* microcomputer and 4 to 124K words of primary memory potentially accessible by all processors. In addition, peripherals such as, teletypes or disks can be present. Also an address mapping device (*Slocal*) is attached to the *LSI-11* bus providing the interface between Cm's in the same cluster.

The *Cm** architecture comprises several clusters connected via *intercluster buses*; each cluster comprising several modules connected through a *map-bus*. Powerful controllers (*Kmap*), which can communicate each with two intercluster buses, provide the traffic control and memory address translations inter and intra clusters, whereas, on the other hand, ensure mutual exclusion on accessing shared data with minimum overhead. However, these maps and their map-buses are the critical shared resources that may lead to 'deadlocks'.

Each *Kmap* contains three major components, the *Kbus* (i.e. a micro-programmed processor providing the interface between the map-bus and the Pmap), the *Pmap* (i.e. a special-purpose processor holding the mapping tables for intra and inter-cluster references) and the *Linc* (i.e. the interface to the two intercluster buses).

In *Figure (I.B.5.3-f2)* is illustrated a three-cluster *Cm** architecture; around 1979, the configuration of the *Cm** comprised 5 clusters of 10 modules each (i.e. 50 modules totally). Theoretically, the architecture can be extended up to 14 modules per cluster as well as in the number of clusters.

On the other hand, this system has some disadvantages; for example, if a module for some reason is lost, then all I/O devices attached to it are

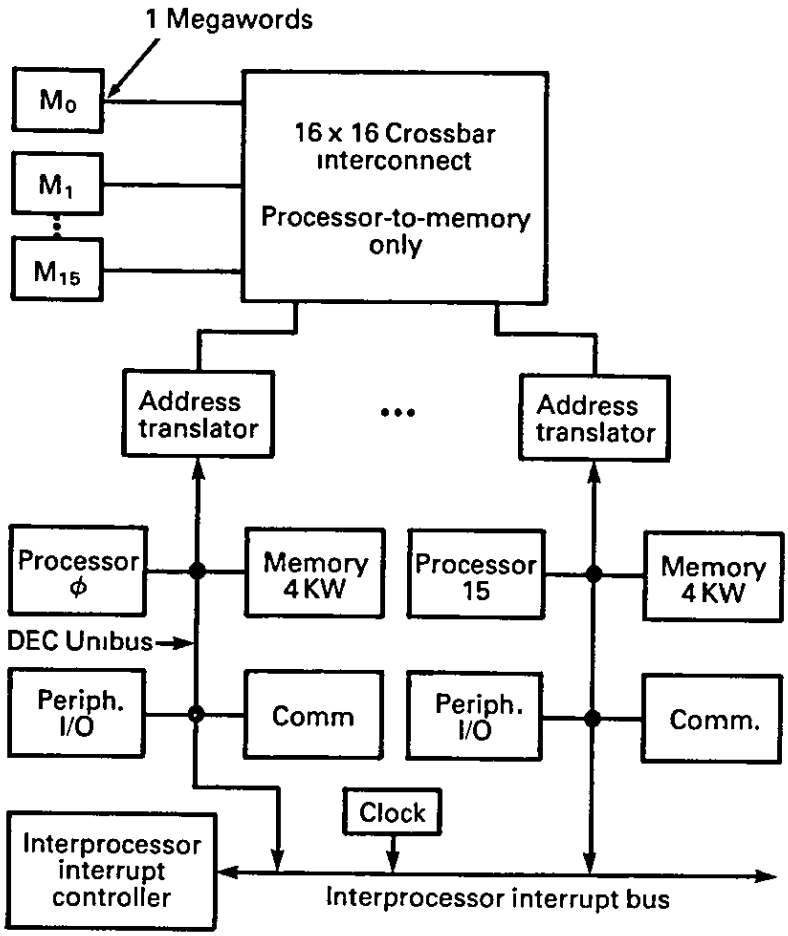


Figure I B 5 3 – f_1 · The *C mmp* Multi – mini Processor

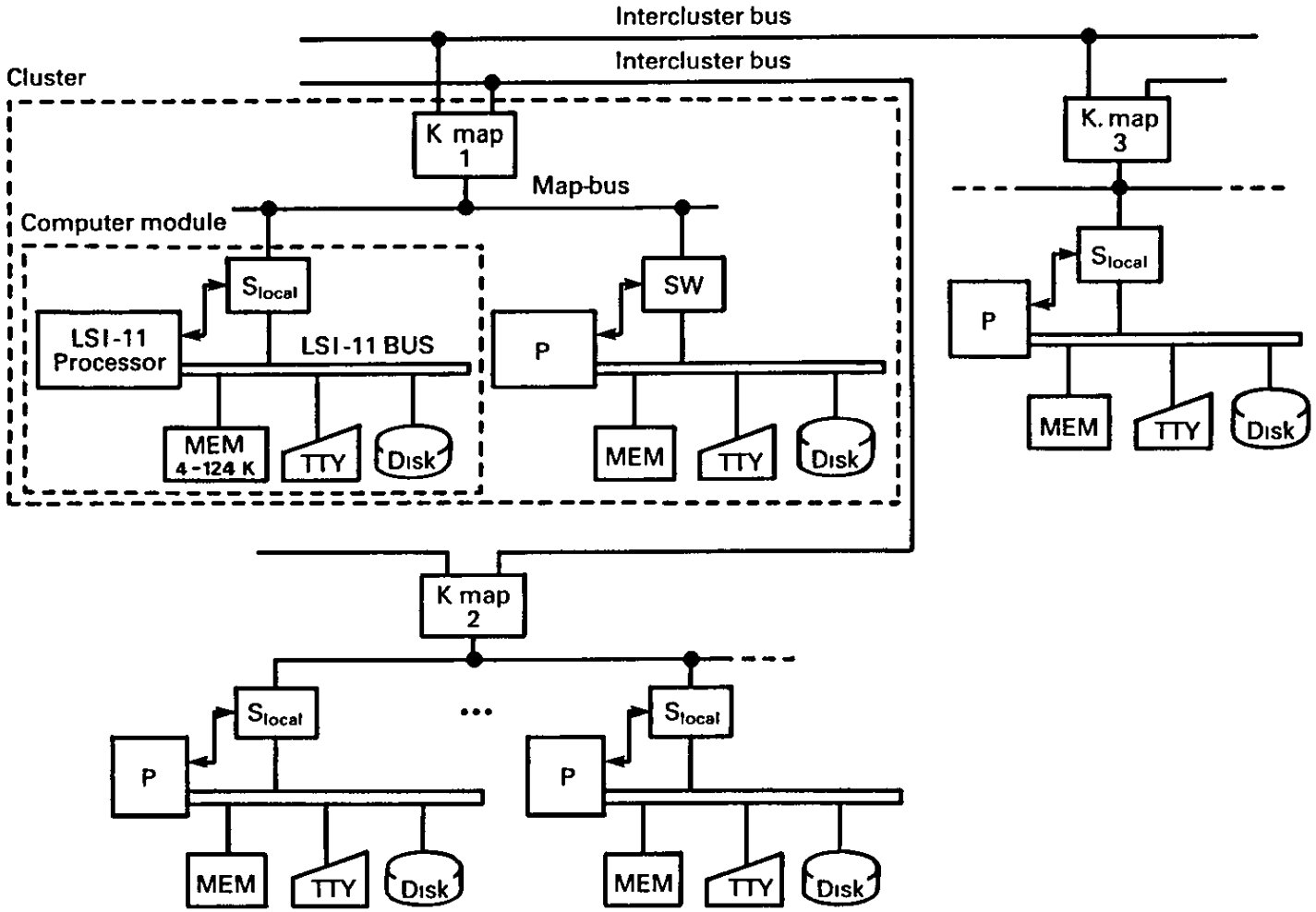


Figure I B 5 3 – f_2 A Three – Cluster *Cm** Network.

also lost; also the data of the local memories are shared by other modules, a fact which causes the 'locking' problem and consequently delays since the messages are transferred in a package-switching mode. Finally, this system is not very suitable for off-line problems which cannot be partitioned into independent subproblems.

In conclusion, and for historical purposes, we mention here an alternative proposal for a *MIMD* architecture, introduced by Flynn, et al, in 1970 [*FLYN70*], which is depicted in *Figure (I.B.5.3-f3)*. According to this scheme, several independent *skeleton* processors, i.e. processors whose arithmetic functions and computational logic have been removed, are interconnected. These functions will be performed by highly specialized high-speed processors, shared amongst the skeleton ones. This sharing is obtained by closely synchronized time-phased switching, thus the resultant system can avoid many of the contention problems associated with shared resource systems. When a skeleton processor requests an arithmetic unit from the 'pool' of the high-speed processors and not one is free, then the request is either placed in a queue or can be repeated until it is finally granted.

The advantage of this proposal is that it can create a Multiprocessor system, without replicating the expensive components of the processors to the same extent that the processors are replicated.

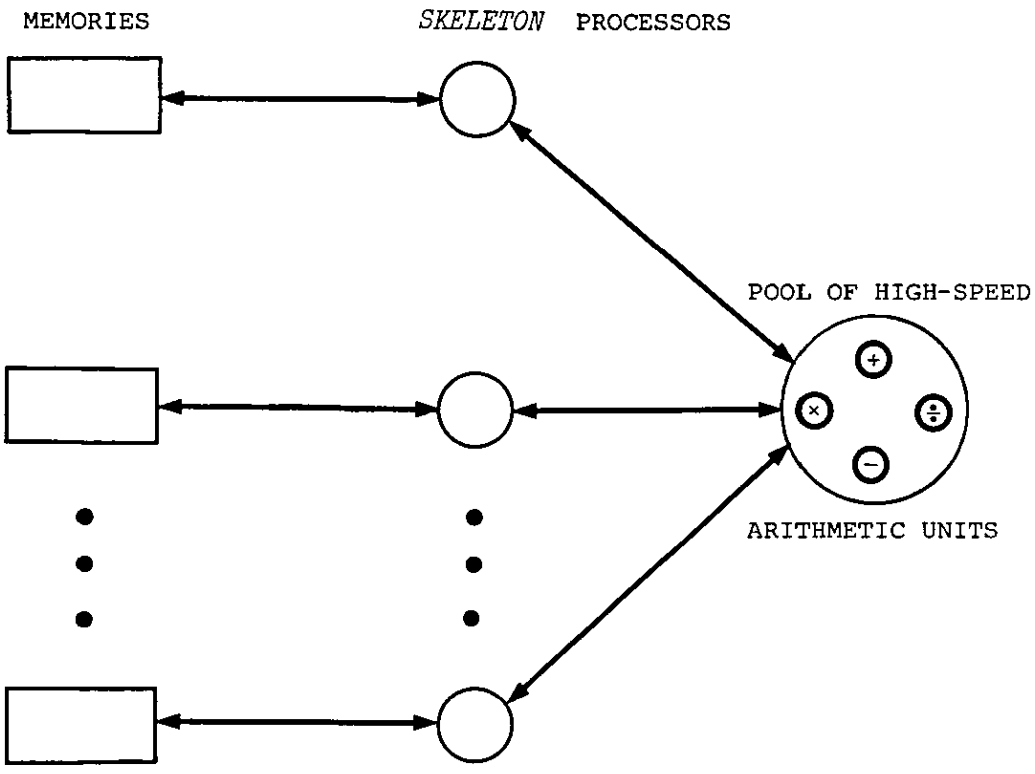


Figure I.B.5.3-f3: A *MIMD* Architecture with *Skeleton* Processors and Centralized Computation Facilities.

I.B.5.3.1: THE INTERDATA DUAL PROCESSOR SYSTEM

The *Interdata Dual Processor*, which can be seen in *Figures(I.B.5.3.1-f1,f2)*, has been developed at Loughborough University. This system first appeared as an Interdata model 55 dual communications processor [*MODE71*]; later on, the I/O processor (*B*), an Interdata model 50 processor, was replaced by a second model 70 processor.

This *twin* Interdata model 70 system provides 32K bytes of *private* (to each processor) memory addressed as bytes [*Ø-(32K-1)*], and a further 32K bytes of shared memory. The *shared* memory, is appended to the private memory of processor *B*, addressed as bytes [*32K-(64K-1)*], thus, actually, the processor *B* had 64K bytes of memory. Each location of the shared memory is referred to by the same address in the two processors.

The *model 70* is a 16-bit processor utilizing 16 registers and operating on an *IBM 360* -like instruction set. Instructions can be 16 or 32 bits long and take 1 or 2μsecs to load from memory. Integers are held as 16 bit halfwords and floating point numbers as fullwords. Floating point operations are implemented in hardware.

The *Interdata Dual Processor* system has the property of *asymmetry*, i.e. the shared memory overheads (*static* and *dynamic*) are not symmetric between the processors. In other words, whereas processor *A* delays by 1 to 1.25 μsecs by the memory bus interface to *B's* direct memory access port, processor *B* experiences no such *static* delay, when they access the shared memory. From the *dynamic* shared memory overhead point of view, if processor *B* accesses the shared or its private memory, processor *A* is 'locked out' of the common memory until the accession is completed; on the other hand, when processor *A* accesses the shared memory then processor *B* is 'locked out' of both, its private and shared memory until the memory cycle (1μsec) is completed. Consequently, both processors experience a delay of one memory

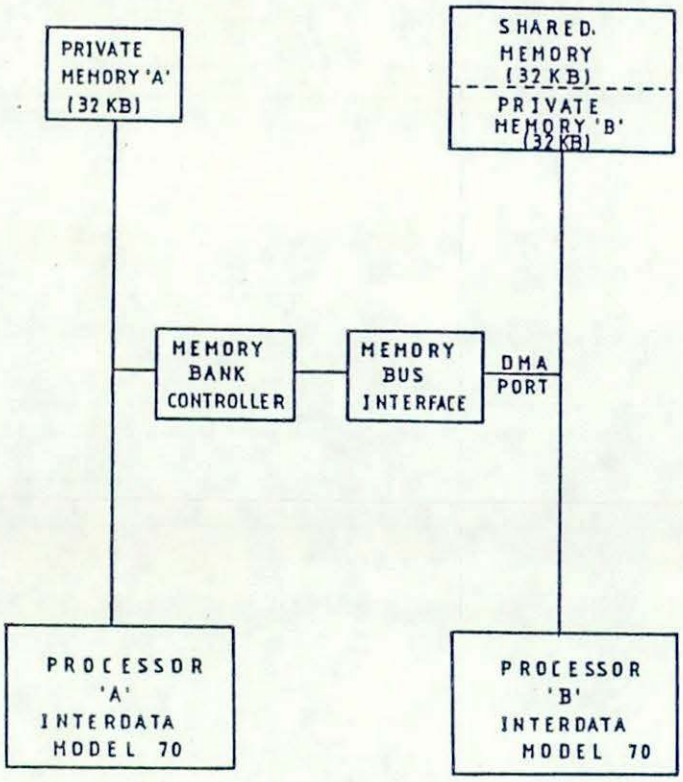


Figure I.B.5.3.1-f1: The *Interdata Dual Processor* System Configuration.

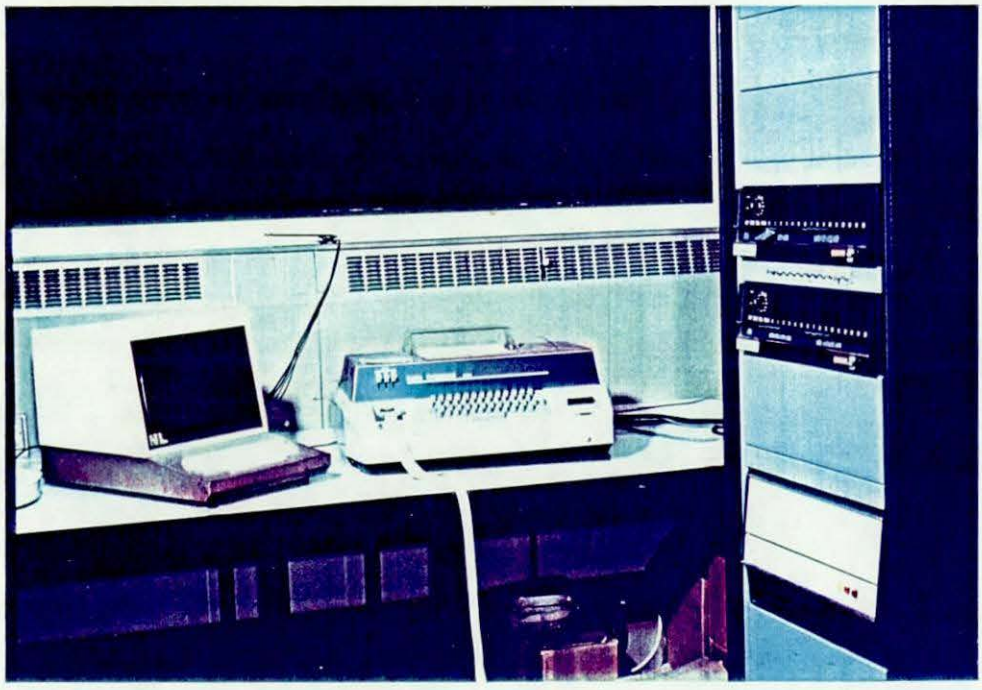


Figure I.B.5.3.1-f2: The *Interdata Dual Processor* System.

cycle (i.e. 1μsec) due to memory contention; in fact, processor *A* reserves the shared memory 0.5μsecs before it can utilize it (due to the memory bus interface logic), which makes the *dynamic* contention more asymmetric. Consequently, in *dynamic* contention, processor *A* delays up to 0.5μsecs while *B* is delayed by up to 1.5μsecs.

The programs can run on one or the other processor, or on both of them, by storing the common data in the shared memory and replicating the code in both the private memories.

In conclusion, the limitations of this system are, primarily the *asymmetry* it possesses and also, the small number of processors, the 64K bytes maximum memory size, the lack of memory protection and the poor quality of the manufacturer's software.

I.B.5.3.2: THE '*NEPTUNE*' SYSTEM

The *NEPTUNE* parallel processing system, yet another type of *MIMD* architecture comprising four *Texas Instruments 990/10* minicomputers, has been developed at the Department of Computer Studies of Loughborough University (see Barlow, et al [*BARL81*], in 1981). Since this system was the vehicle utilized to implement a significant part of the research presented in this Thesis, we discuss in more detail some of its specific hardware and software features. The related programming concepts and the performance measurements of the system will be presented in *Chapter II*.

The *Physical* current organization of the *NEPTUNE* system, with an up-to-date potential number of connected terminals[†] and storage capacity, as well as the actual *NEPTUNE* system itself, are shown pictorially in *Figures (I.B.5.3.2-f1,f2)*.

The system comprises *five* linked buses (*TILINES*), four of which are

[†] *The up-to-date system, theoretically, can accept four (at a maximum) memory mapped VDT's and nine RS232 standard terminals. Hatfield Polytechnic possesses one of them via the Gandalf network.*

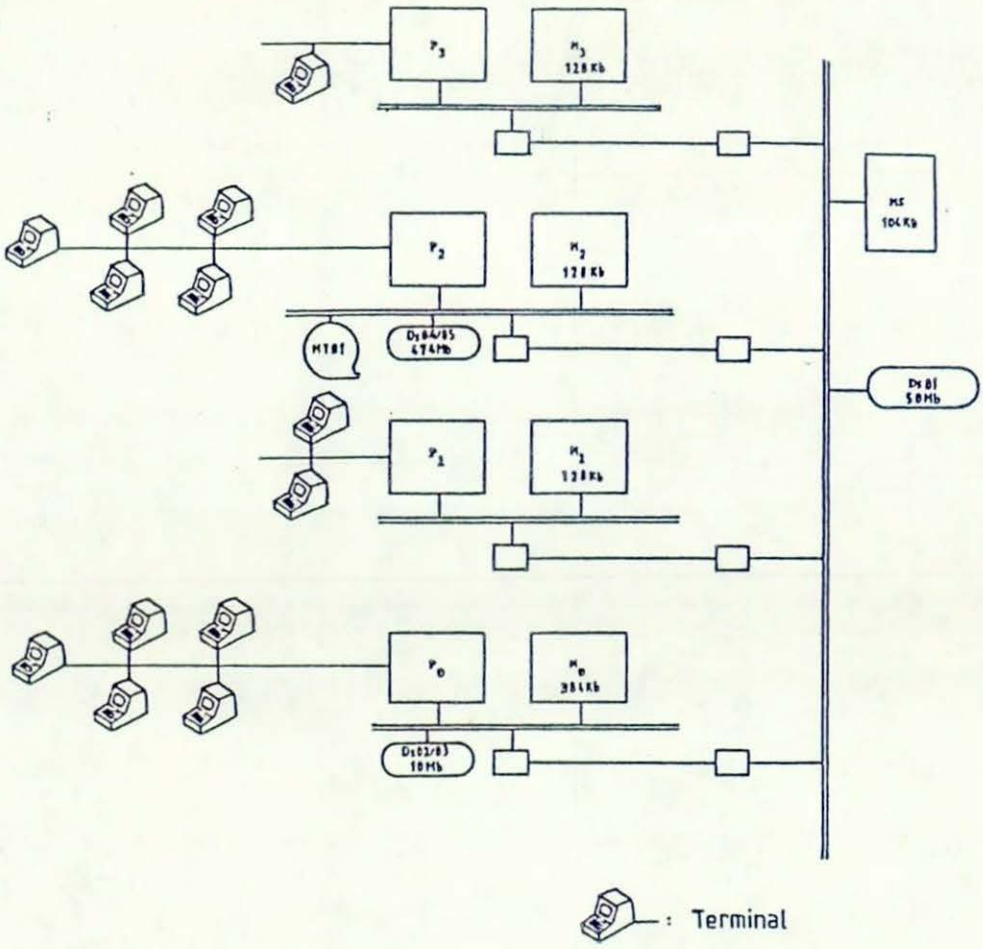


Figure I.B.5.3.2-f1: The Current *'NEPTUNE'* System Configuration.

Figure I.B.5.3.2-f2: The *'NEPTUNE'* System.

connected like *local* buses to the corresponding processor. Each processor, via its local *TILINE*, can access its own (*private*) memory of at least 128K bytes capacity (a capacity of up to 512K bytes is envisaged, e.g. the memory of processor $\emptyset$ has now been increased to 384K bytes). In addition, processor $\emptyset$ has a 10M bytesdisk drive on its local *TILINE* (actually it is two disks, one fixed and one exchangeable, each of 5M bytes capacity); the processor 2 has a controller with 474M bytes Winchester disk drive as well as a magnetic tape streamer attached to it.

Each of the four local *TILINEs* is connected, via a *TILINE* coupler, to a fifth shared *TILINE*, on which 104K bytes (rising to 128K bytes) of memory and a 50M bytes disk are attached. This memory can be addressed by all processors, consequently a minimum, not normally contiguous, addressable space of 232K bytes is available. In addition, the 50M bytes disk can be accessed by each processor with disk interrupts being transmitted to each of them.

In a brief reference to the processors' specific characteristics, these are identical in many hardware features, though they present some differences in their memory accessing speeds. More specifically, the time for each processor to access its *local* memory is ~0.6μs, whilst the *excess* access time to the *shared* memory is 0.81μs, 0.52μs, 0.71μs and 0.72μs for processors P_0,P_1,P_2, and P_3[†], respectively. Consequently, the total *shared*memory access time is the sum of both access times to the *shared* (*excess*) and *local* memory made by each processor; in addition, the relative speeds of processors P_0,P_1,P_2,P_3 are 1.000, 1.037, 1.006 and 0.978[†], respectively, a fact which also contributes to a reduced efficiency and decreases the performance measurements of an algorithm with synchronization.

[†]*In an attempt to provide as much as possible up-to-date information about the system, we must mention that the speed measurements may slightly vary now, which is unofficially verified by test runs carried out by the staff supporting the system.*

From the *logical* organization point of view and during normal use, this system operates like four individual processing systems. Any processor, at any time, of course subject to availability, can request storage in the shared memory and once allocated this storage it treats it as if it was a slower local memory. A small shared memory area on top, is mainly utilized to store the data structures used in the management of the shared resources (including the shared memory itself) and to provide the inter-processor communication.

Any parallel program to be implemented on this system logically consists of two parts, one which contains the *program code* and the *local variables* and another containing the *shared variables*; these two parts reside in two different segments[†], which is ensured by the commands used to generate the parallel programs. Now, in the case that a processor receives a request to execute a parallel task, its first action is to claim shared memory area and once allocated, to load into that space the segment containing the shared variables; subsequently, the *management* area (in the shared memory) is set to contain pointers to that shared segment and tasks are activated in other processors with sufficient information for them to execute the requested program. On the other hand, the segment(s) containing the local variables are loaded into the private memories of the corresponding processors, whilst the other processors, except the initiator, *link* into that shared segment which is residing in the common memory.

From the *Operating system's* point of view, each processor runs under the powerful *DX10* uniprocessor operating system, which is a general-purpose, very sophisticated multi-tasking system. It features an effective file management package, which includes support for multi-key indexed files. Modifications have, however, been underway for the *DX10*, to produce a *new*

[†]*The 990/10 hardware allows a program to exist on up to three segments.*

mark of the Operating System (*DX10 Mk 3.5*) along with the instalment of a new hardware (memory, hardware floating point, resource management) and the development of some new facilities (e.g. a new preprocessor for use on the *VAX*, a file transfer mechanism between *VAX* and *NEPTUNE* etc.). The files that are stored on the shared disk by *DX10*, are available to tasks running on all processors simultaneously and consequently a *coordination* procedure is required to allow files to be *created*, *opened*, *accessed*, (i.e. read and written) as well as *deleted*, by more than one processor. 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. when two tasks on different processors open a file, one to read and the other to write, the reader is informed about the changes made only when the writer closes the file) are still restricting the simultaneous accesses of a single file.

From the user's point of view, the 'System Command Interpreter' (*SCI*) provides the user interface to the *NEPTUNE* system. The *SCI* provides several ways in which commands may be issued. At the simplest level, a sequence of *Menus* is displayed on the terminal's screen, driving the user through a coherence of command classes[†] and eventually reaches a list of commands. The command parameters required can be either typed following the command or wait for *SCI* to prompt them; in any case it is *SCI's* role to perform a check on the given value. The commands can be implemented either as functions of the *SCI* (e.g. supervisor calls), or as tasks running under the Operating System (e.g. the compilers, the utility programs).

DX10 provides and supports a *tree-structured* filing system in the form of a sequence of *Directories* starting from a specific disk pack or *volume* maintained independently. The specification of the files depends on the volume they reside on; in other words, if they reside on the system volume,

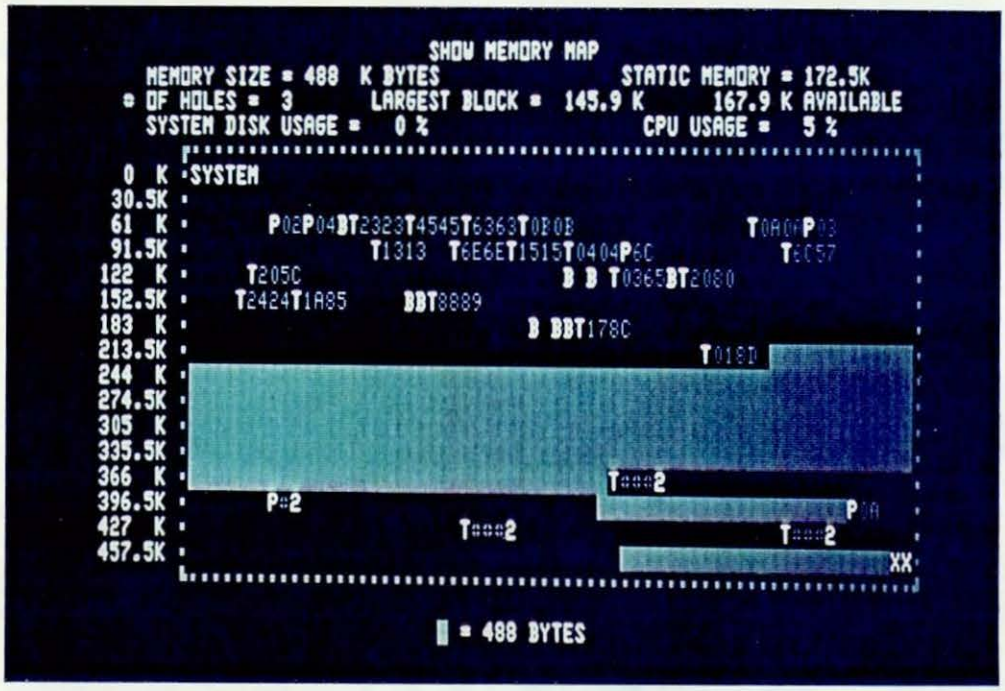
[†]*The Menu sequence can be skipped by typing the command name directly.*

the volume name can be omitted, whilst in any other case a full filename specification is required starting from the volume name.

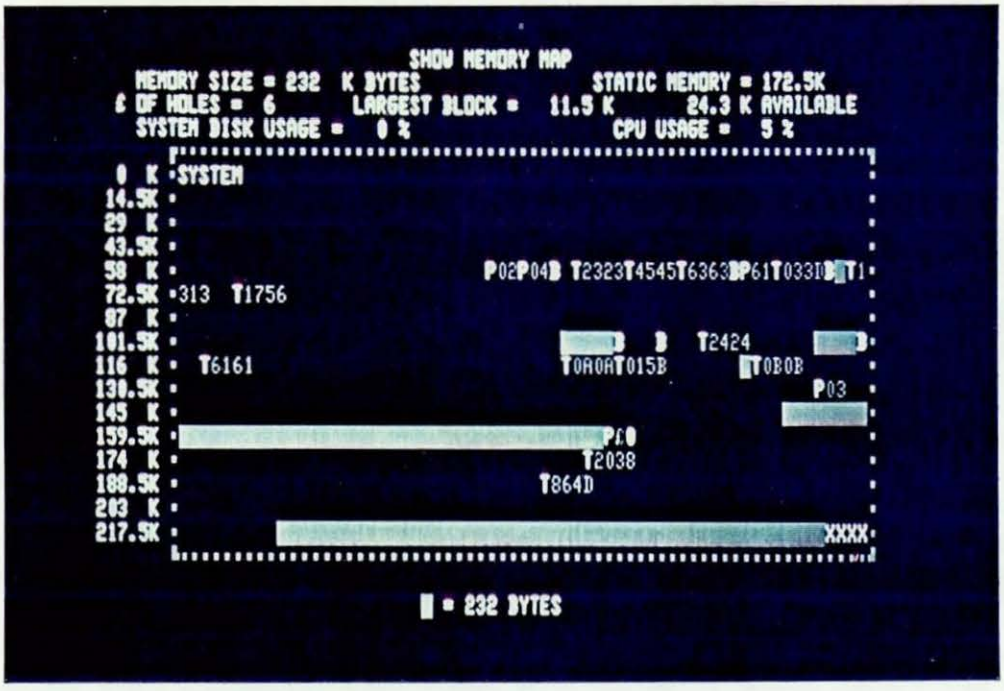
For simplicity purposes *synonyms* may be utilized, which are especially useful in the case of long character strings; for example a Directory *VOL1.DIR1.DIR2* can be replaced by the synonym *DIR*, which means that all files in this directory can be referred to as *DIR.filename*. Again, it is SCI's role to evaluate the given synonym. A variety of commands concerning the file management and memory allocation [see *Figure (I.B.5.3.2-f3)*] are available in the *SCI*. Generally, for file and Directory operations we refer to the reference manual [*Texas Instruments,II&IV*] the commands concerning file editing and running along with some other facilities will be discussed in *Chapter II*.

In the Texas Instrument 990/10, the *foreground* and the *background* features are available during the execution of a task. The difference between these two features is that whilst the background is a multi-tasking management environment, the foreground can be *owned* by one only user for only one task to *run*, at any time, suspending the *SCI*. However, background tasks should not involve I/O with the terminal and commands are available to inspect the background status, since the *SCI* is still running and is available to process user requests.

Since *NEPTUNE* is a research system and consequently as such, is subject to frequent changes and extensions, it is bound from time to time to experience some malfunctioning problems. The normal symptom of a malfunction is when one or more processors *fail*, indicating a fault (the front panel's 'FAULT' light goes on, although we may have a fault without this indication); this malfunction is known as a system *crash*. To reload the system there is a manual procedure on the front panel of each processor, concluding with *IS* ('Install System') command, which will give back the message 'Initialization Complete'.



(i) - *[Processor Ø]*



(ii) - *[Processor 2]*

Figure I.B.5.3.2-f3: Pictorial Representations of the Memory Allocation to Various Tasks.
[Obtained utilizing the 'Show Memory Map' - *SMM* command].

To avoid undesirable situations which can be arisen in a variety of ways (e.g. disk corruption), the system provides a *dumping* facility, in a short or long[†] term basis, of all files on the disk packs *PARUSER1* (i.e. Parallel User source files) and *FIXED* every night, except Sunday; installed programs and temporary files are not preserved.

Finally, another important software extension made to this system, is the development of a *PASCAL* version[‡] allowing communicating programs; also a project to offer a *PASCAL-PLUS* implementation is underway, as is one for a multiprocessor simulator.

In conclusion and anticipating the future work to be carried out in the Department of Computer Studies, at Loughborough University, a group has been actively researching in the area of *MIMD* systems proposing a different configuration (similar to *Cm**) for the construction and development of a larger and more powerful system, possibly comprising 16 processors.

[†] *On a separate disk, normally for large and important files which will need to be kept for a month or more.*

[‡] *In addition to the FORTRAN based parallel software.*

I.B.6: A General Review Of Multiple Processor Systems' Principle Motivations

In reviewing some aspects of parallel systems, one must recognize that technology is considerably ahead of software and architecture, a fact which affects all the motivating characteristics mentioned below. This is because modern electronic components are remarkably reliable, whilst the complex software systems are bound to introduce errors into the programs since they are subject to continuous maintenance, modification and development. Currently there is no Operating System known which controls tasks and processes as easily as block languages handle functional blocks. Probably, *Petri nets* or related nets, as well as *flow* schemes and special hardware, which we shall discuss in *Chapter III*, are a step in the desired direction.

The motivating goals for multiple processor systems development projects, and it is somewhat remarkable, they have remained basically unchanged since the earliest days of digital computer systems. The most important of these long-sought-after motivating goals are, an increased system *productivity* (i.e. greater *capacity*, shorter *response* time, increased *throughput*), an improved *flexibility* and *reliability*, an improved ability to *share* system resources, and the *ease* of system expansion. Since these goals are not expressed in absolute numbers, it is not surprising that they continue to apply albeit phenomenal advances have been made in many of the areas, such as *speed*, *capacity*, and *reliability*.

The increase of system *productivity* was envisaged that will be obtained by increasing the number of processing elements in the configuration, since the speed of logic circuitry, with continuous enhancement, was approaching its physical ultimate. In addition, the utilization of more processing elements also greatly increased the system potential *flexibility*. This flexibility may be used to increase the *reliability* of the system,

but it has considerable implications for the allocation of the system available resources to the workload. More specifically, when more than one program may be run simultaneously, there is a greater opportunity for *sharing* critical resources like large sets of data.

One of the most significant motives for developing a multiple processor system is to attain a greater measure of *reliability*. However, the term reliability is often very 'loosely' applied and computer systems designers sometimes overlook the fact that computer reliability combines two related, but distinct, aspects required by different applications to different degrees; these are the system *availability* and the system *integrity*.

The requirement that the processing capacity of the system should remain available to all albeit something is going wrong in the system, defines the *availability* of the system (e.g. in computer controlled telephone exchanges, airline reservation systems, etc.).

On the other hand, the requirement that the results produced by the system should be always 'correct', defines the *integrity* of the system (e.g. in a banking system, etc.); in other words, this feature should be considered as a requirement to 'protect' the information content of the system. However, the cost of *integrity* is greater (as greater as is its importance) than that of *availability* and consequently the cost of the entire system that needs both is very high.

As an *Epilogue* to this *Chapter*, it seems that every new major system concept or development (e.g. *multiprogramming*, *multiprocessing*, *networking*, *distributed processing*, etc.) has been presented as *the answer* to achieving all of these motivation goals. Also we must underline the fact that, by no means, we have referred to all computer architectures which have contributed to the notion of parallelism (e.g. the *EGPA*, *CII IRIS 80*, *Fairchild SYMBOL 2R*,

CYBA-M, etc., computer architectures have not been mentioned); simply we have referred to a selection of some well known commercially implemented systems.

In conclusion and in order to bring up-to-date the classification of multiple processor systems, we must mention some other classifications presented by Hockney and Jesshop [*HOCK81*] and by Händler [*HAND82*]. Both of these classifications have introduced an innovative mnemonic(!) sort of structural notation; the first one in the form of *chemical* formulae[†], whilst the second one (i.e. *ECS* - 'Erlangen Classification System') introduces a simple, lucid but rigid triplet as a characterization for basic structures, as well as the operations '+','*', and 'v' in order to make compositions of the available structures.

[†]*The 'Multiprocessors' category has been entirely omitted(!).*

CHAPTER II: PROGRAMMING TOOLS AND ALGORITHMS TO EXPLOIT THE PARALLEL HARDWARE POTENTIAL

SECTION A: PROGRAMMING LANGUAGES AND CONCEPTS FOR PARALLEL PROCESSING

II.A.1: INTRODUCTION

II.A.2: *'CONCURRENT'* PROGRAMMING LANGUAGES MOTIVATIONS AND TRANSFORMATIONS OF SEQUENTIAL PROGRAMS INTO PARALLEL PROGRAMS

II.A.2.1: A LEVEL-DETECTION OF PARALLELISM

II.A.2.2: EXPLICIT PARALLELISM DETECTION APPROACH

II.A.2.3: IMPLICIT PARALLELISM DETECTION APPROACH

II.A.3: PROGRAMMING CONCEPTS OF THE LOUGHBOROUGH MIMD MULTI-PROCESSING ARCHITECTURES

II.A.3.1: THE USER INTERFACE TO THE *'NEPTUNE'* PARALLEL PROCESSOR SYSTEM

SECTION B: ASPECTS OF PARALLEL ALGORITHMS DESIGN AND ANALYSIS METHODOLOGY

II.B.1: CONSTRUCTION PRINCIPLES FOR EFFICIENT PARALLEL ALGORITHMS

II.B.2: SCHEMES AND TECHNIQUES TO DESIGN ALGORITHMS TO MAP ONTO SIMD AND PIPELINED VECTOR COMPUTER ARCHITECTURES

II.B.2.1: PARTICULAR CONCEPTS AND PERFORMANCE FEATURES OF THE *'DAP'* SYSTEM

II.B.3: FUNDAMENTAL ALGORITHM STRUCTURAL CONCEPTS TO EXPLOIT THE POTENTIAL OF MIMD COMPUTER ARCHITECTURES

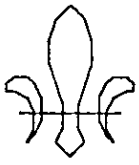
II.B.3.1: MULTIPROCESSORS PERFORMANCE ANALYSIS CHARACTERISTICS AND RESOURCE PROVISIONS OF THE *'NEPTUNE'* PARALLEL PROCESSOR SYSTEM

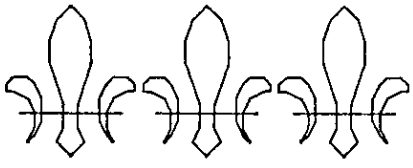
Chapter II

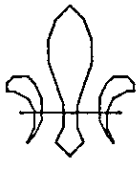
Programming Tools And Algorithms To Exploit The Parallel Hardware Potential

SECTION Å

PROGRAMMING LANGUAGES AND CONCEPTS FOR PARALLEL PROCESSING







'We dissect nature along lines laid down

by our native language...

Language is not simply a reporting device

for experience , but

a defining framework for it.'

Thinking in Primitive Communities

Benjamin Whorf 1897–1941

II.A.1: INTRODUCTION

Although it is possible and useful to separate the system software structure from the hardware structure, however, since a serious mismatch between them is likely to lead to an ineffective overall system, the state-of-the-art advances - in particular, anticipated advances generated by *LSI* - have given a fresh impetus to research in the area of parallel systems software.

From the *hardware-software* relation point of view, all the parallel architectures mentioned in *Chapter I* and the corresponding system software organizations utilized to manage parallelism, serve a purpose on which their configuration was based in the first place. In other terms, for some architectures the hardware structure has concealed the parallelism itself, as in the pipelining, whilst for others the system software designer had to decide whether to reveal (or not) the parallel architecture to the user, or how to exploit the hardware capabilities in the system software itself.

In consequence, if the overall system *reliability*, imposed by the replication of various components to provide a high probability of one remaining operational at all times, the *ease* of implementing the system,

as well as the overall system *throughput*, were considered to be the system configuration reasons, then the concealment (from the application programmers) of the parallel hardware facilities, giving the appearance of a uniprocessor system, would be appropriate.

On the other hand, if the system configuration reason was to provide a higher execution speed within a single program, then the concealment of the hardware parallelism would be quite counterproductive, although its effective utilization through the system software is quite a difficult task.

As a general rule, the *MIMD* architectures are more likely to have been configured for overall reliability, ease and throughput, whilst the *SIMD* architectures are preferably configured for high execution speed on particular classes of problems.

Any parallel system is considerably more difficult from the programming point of view than a uniprocessor. In the case of many *SIMD* systems, where the system software simply ignores the parallel architecture allowing the application programmer to benefit from a parallelism free of system software overhead, it causes a heavy burden of programming; this may lead to a longer development time for programs which can be eventually proved of low potential parallel benefit (e.g. low execution speeds).

The alternative solution is a system software 'partially' concealing the parallelism; for example, compilers translating vector or array operations directly into appropriate hardware operations, although, again, a maximum hardware speed is not normally obtainable and the development times are also decreased.

More specifically, the term 'parallelism' can be applied to the

system software in two different ways, the *actual* and the *hierarchical* parallelism. With reference to a single program (job), the term *actual* parallelism can be applied at several levels as we shall discuss in a subsequent paragraph; to the contrary, the term *hierarchical* parallelism refers to the task (process)† parallelism which can exist at several levels within a hierarchy of levels. For example, the 'statements' within a program can be characterized as *first*, *second*, *third*, or etc. level, if they reside in the main program, in a subroutine called by the main program, in a subroutine called by the previous subroutine, etc., respectively. In *Figure (II.A.1-f1)* we can see the way a sequentially organized program can be represented by a hierarchy of levels.

After the level-by-level analysis of a sequentially organized program the primary consideration to exploit parallelism resides on the identification of those tasks which can be executed in parallel. The deterministic approach to this problem can be made from two directions, the *explicit* and the *implicit* one, which we shall present in subsequent paragraphs. The information obtained by any of the above approaches must be sent to and utilized by the Operating System, since an efficient resource utilization is the prime consideration at this point.

In *Figure (II.A.1-f2,a)*, a sequentially organized program is depicted containing a number of τ_i tasks. If the tasks τ_1 and τ_2 can be executed in either order and still leave τ_3 unaffected, then parallelism can be said to exist between tasks τ_1 and τ_2; their parallel execution is illustrated in *Figure (II.A.1-f2,b)*.

---

†*A task can be generally defined as a self-contained part of a computation, which without any further additional inputs can be carried out to its completion - e.g. a single statement or a group of statements.*

In conclusion, we must underline that this 'commutativity' condition is necessary but not sufficient for parallel processing, since there may exist processes with similar independence in the execution order but not processable in parallel (e.g. the inverse of a matrix A)[†].

Certainly other complications may arise due to hardware limitations (e.g. accesses to the same memory, etc.)[‡]. Bernstein [*BERN66*], in 1966, pioneered investigating the deterministic conditions (*implicit* approach) for the parallel execution of two tasks (see *par. -II.A.2.3*). These conditions were sufficient to guarantee 'commutativity' and 'parallelism' between tasks, although Bernstein had shown that there did not exist algorithms for deciding the presence of these factors.

[†] *The inverse of a matrix A can be obtained in three distinct processes; τ_1: obtain transpose of A, τ_2: obtain matrix of cofactors of the transposed matrix, τ_3: divide result by determinant of A. The τ_1 and τ_2 processes can be equally well commutated in the execution order, they cannot though be executed in parallel.*

[‡] *Dijkstra [DIJK65], Knuth [KNUT66], and Coffman, Muntz [COFF69] developed efficient scheduling procedures for using common resources.*

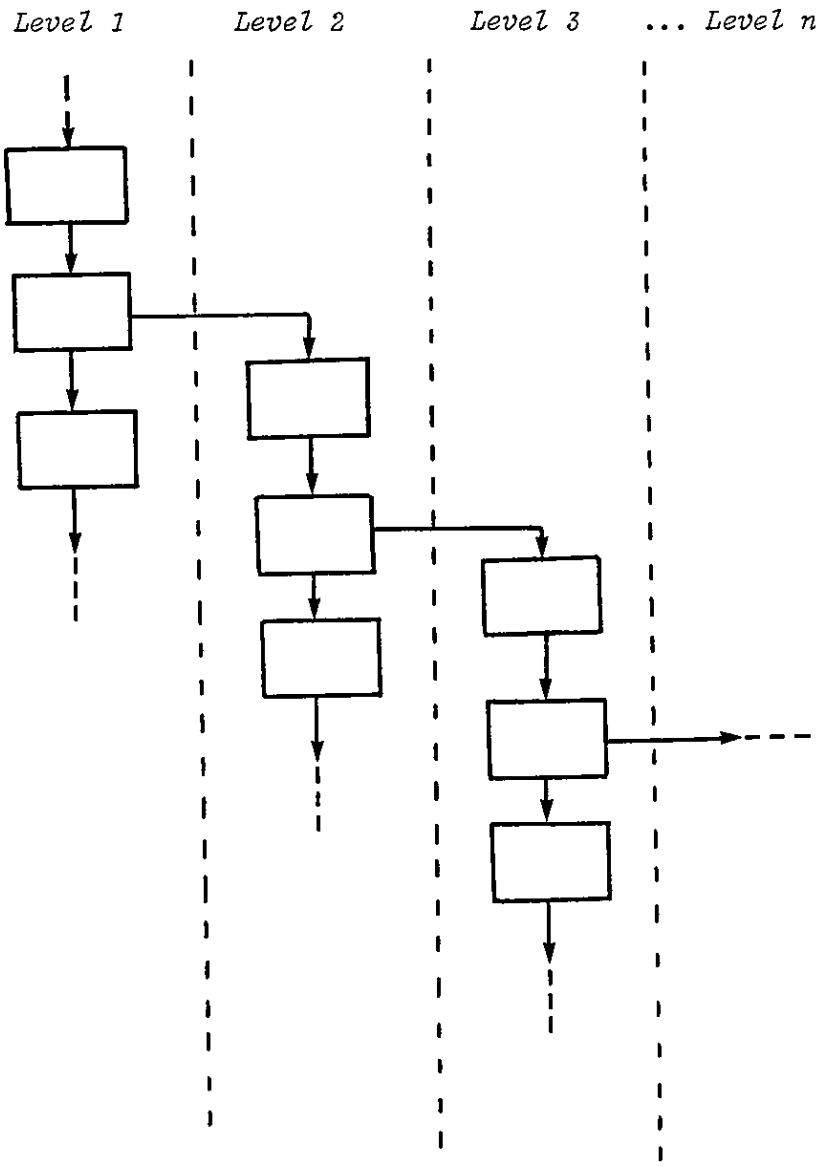


Figure II.A.1-f1: Hierarchical Representation of a Sequentially Organized Program (Each block within a level represents a single task).

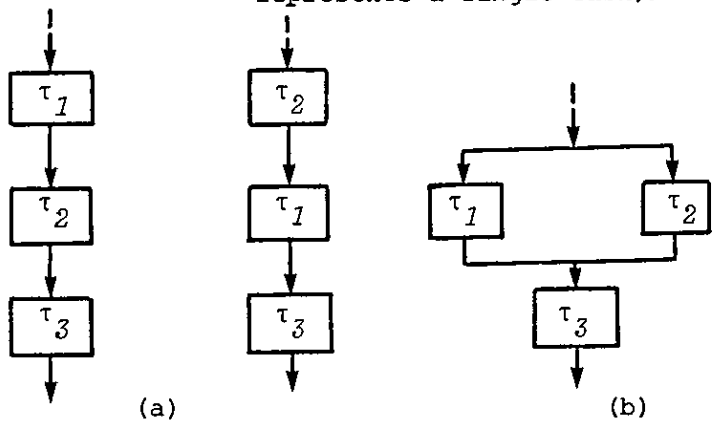


Figure II.A.1-f2: Sequential and Parallel Execution of a Task.

II.A.2: 'CONCURRENT' Programming Languages Motivations And Transformations Of Sequential Programs Into Parallel Programs

The majority of the commercially available parallel systems were, until quite recently, basically sequential systems where 'parallelism' occurs in the form of vector instructions or pipelined streams at lower levels and quasi-usual programming languages at higher levels.

The reasons for this status are quite simple and reside on the fact that the parallel system manufacturers when building their systems found a vast software background written in *FORTRAN*, the language on which computer scientists argue if it was 'invented' by God or devil. Consequently it was very profitable for them to consider *FORTRAN* as *the* language for their system, and just improve its absolute sequential character by introducing 'extensions' to describe the *obvious* parallelism† available and developing optimizing compilers to match (if possible) the hardware level control mechanisms.

However, despite the fact that an increased interest in 'parallel' languages has arisen primarily due to the theoretical potential of parallel hardware, the real motivation for 'parallel' languages should come from the programmer's needs. High level languages have developed precisely because they provide concepts relevant to the programmer independent of machine architectures.

When designing new programming languages a 'key' problem about them is the representation of programs. Any programming language primarily aims to serve two main purposes; first, to express the programs in an

---

†*Since FORTRAN is a sequential language the only place where parallelism can be found out is the DO-loop statements, within which actions are applied to objects several times.*

abstract representation and, secondly, to instruct the system to execute a program. Typically, after the representation of a program in a high level language a transformation via a compiler takes place, into another representation executable by the hardware. The basic problem is the optimization of these transformations in terms of certain measures, e.g. size, execution speed and potential parallelism.

A plethora of 'parallel' programming languages have been proposed for different types of systems. For example, Coulouris (see Iverson [*IVER62*]) investigated the implementation of a high level language system for the array processing language *APL*. Also, *TRANQUIL* (see Abel, et al [*ABEL69*]) an *ALGOL*-like language, and *ACTUS* (see Perrot [*PERR80*]), have been designed to exploit parallelism in algorithms to be implemented on Array Processors, such as *ILLIAC IV*. Per Brinch Hansen [*HANS77*], developed the *CONCURRENT PASCAL*[†], at the California Institute of Technology from 1972-75, partly supported by the National Science Foundation. Also, we should not miss out the work carried out on the *ICL DAP* system (see Flanders, et al [*FLAN77*]), which allowed access to the primitive machine interface[‡], embodying a high degree of parallelism to be exploited by the *DAP FORTRAN* language developed by Flanders in conjunction with *ICL* (see Flanders [*FLAN82*]).

The current parallel programming representations may be characterized as inadequate, since they suffer from a lack of flexibility which makes decomposition, optimization and translation difficult. This problem would

---

[†] *Since then, other extended versions of PASCAL, such as the PASCAL-PLUS (see Welsh and Bustard [WELS79]) and the 'dynamic' PATH PASCAL (see Dowsing and Elliott [DOWS84]) (processes and objects can be created at run-time, unlike in CONCURRENT PASCAL), have been introduced.*

[‡] *The use of the primitive level hardware interface, often referred to as 'microcode', provides a 'virtual' system, capable of executing a specific high level language (see Iliffe [ILIF82]).*

become even worse in a 'Very Large Scale Integrated' (*VLSI*) environment (see *Chapter III*), since the existing programming languages would not be able to efficiently utilize the much greater opportunity for parallelism offered by the *VLSI*; the rise of special-purpose chips and architectures also requires more flexible representations for programs.

The sort of merits that scientists are seeking for, we think can be obtained from the *applicative* or *functional* language representations[†], in which a computation is expressed as a function evaluation. This type of representation offers a very small number of basic concepts, minimized complexities, a lack of central state or global environment (due to the fact that functions are dependent only on their inputs), and a very highly parallel nature thus being suitable for the *VLSI* environment.

Certainly there is a number of important, yet unsolved, problems to overcome, if functional language representations are to succeed, such as, the ability to match existing and future architectures, human engineering qualities, effectiveness in terms of parallelism potential, utilization and representation of data structures, etc.

Finally, to come up-to-date with the technological advances, we must mention the forthcoming, so-called, *transputer* (see also *Chapter III*) and the *OCCAM* language (see May and Shepherd [*MAYS84*]), which are designed by *INMOS*; *OCCAM* was initially designed as a concurrent programming language, and is the lowest level at which the *transputer* can be programmed, in effect an assembler language for parallel programming. The theoretical basis for this language was worked out by Professor C.A.R. Hoare of the Program

[†] *Some of the most well known 'functional' languages are, the Formal Functional Programming language (see Backus [BACK78]), the Kent Recursive Calculator language (see Turner [TURN82]) and the Saint Andrews Static Language (see Turner [TURN79]).*

Research Group at Oxford University. He developed the concept of 'Communicating Sequential Processes' (*CSP*) [*HOAR78,HOAR80*] for the U.S. Department of Defence, but they chose *ADA* instead as their new real-time language. They, then, added the primitives from *CSP* to *ADA*, making the language even more unwieldly, although *CSP* was supposed to be a simple, lean-and-hungry language.

Another recently developed parallel language is the *CONCURRENT EUCLID* (see Cordy and Holt [*CORD83*]), a modern *PASCAL*-based programming language to be utilized in the 'Network Access Controllers' (*NACs*) of a functional system called *Hubnet* (see Lee and Boulton [*LEEB83*]).

From the actual programming point of view, in order to program in a way to allow 'concurrency', one needs to reveal natural dependencies of the subparts, but not to introduce new dependencies by overspecifying the problem.

This can be done by the programmer himself (*explicit approach*) indicating the tasks within a computational process which can be executed in parallel by means of additional instructions in the utilized programming languages. However, the intrinsic complexity of a description of concurrency has unavoidably led to investigate automatic transformations (*implicit approach*) of sequential programs into parallel ones. Much research has been devoted to this topic; Baer [*BAER73*] and Kuck [*KUCK75*] have surveyed all the various proposed methods.

In conclusion, these methods generally can be classified into two complementary categories. The first, refers to *local* transformations, attempting to underline the inherent parallelism of a certain task, depending, either on the semantics of some particular operators: *FORTRAN DO*-loops [*KUCK75*], commutativity, distributivity and associativity of

arithmetic operations [*BAER73*], or on the semantics of data structures (simultaneous accesses to parallel hyperplanes of an array), see Lamport [*LAMP75*]; the second category, refers to *global* transformations according to which the sequential programs are syntactically analyzed and their control structure is completely modified, thus allowing the parallel execution of instructions utilizing distinct variables.

The *explicit* and *implicit* parallelism detection approaches will be discussed in greater detail in paragraphs (*II.A.2.2*) and (*II.A.2.3*), respectively.

II.A.2.1: A Level-Detection Of Parallelism

As was explained by the various architectures of existing parallel computers, parallelism can be achieved in a variety of ways. Attempting to summarize all these possible known ways of achieving parallelism and categorize them into several distinct levels, we obtain:

a) *Job Level* → between jobs;
→ between phases of a job;

b) *Program Level* → between parts of a program;
→ within *DO*-loops;

c) *Instruction Level* → between phases of instruction execution;

d) *Arithmetic and Bit Level* → between elements of a vector operation;
→ within arithmetic logic circuits.

The provision of a correctly balanced set of replicated resources, coming under the general classification of functional parallelism, is the main significant requirement of a computer architecture in order to

allow parallelism at the highest (*Job*) level. In this respect it is of great significance that the overall activity in all parts of the installation be monitored, in order to assist in identifying bottlenecks and to add or remove any resources according to the specific problem demands. The maximization of the processing rate of *Jobs* is the objective of the system at this level.

In a simpler analysis, each job can be considered as a set of several sequential *Phases*, each of which requires a different system program and resources. These phases could be considered to be the *input* program source code, *read* from a disk or tape, the *compiling* of this code into object code, the *linking* of the object code with any needed library subroutines, the *execution* of the resulting module and finally the *print out* of the result files.

The first and the last phases (I/O operations), compared to the execution phase, are considerably slower, so all large computer installations offer several I/O channels or peripheral processors in order to perform I/O in parallel with the program execution, providing a battery of disk and tape drives.

On the other hand, individual installations use a different number of processors for the execution of the programs, but in all cases only one computer program, the so called *Operating System*, exists to control the flow of the work through the system, organizing the sharing of the system resources amongst the various jobs.

In the case that the installation has only a single processor and in the *fast* memory reside many programs for execution, then the execution is sequential, starting with the first program; by the use of an interrupt procedure a dynamical overlapping process can occur with the *suspension*

of the *active* program (e.g. demand for I/O), and the execution of the next program in the queue commences. The operation of the I/O (e.g. read from the disk or tape) is then initiated in the channel; the control for the execution of the *suspended* program is regained when the other program(s) are similarly *suspended* and of course the former program has gone out of this state.

In the subsequent lower (*Program*) level of parallelism, we may have a program including *Sections* of code quite independent of each other, thus being possible to be executed in parallel on different processors, in a multiprocessor environment (e.g. a set of linked processors like the *NEPTUNE* (*MIMD*) parallel system).

A logical analysis of the source code reveals some of the independent sections, since for others some constraints still exist; for example, a data dependency which cannot be revealed before program execution. Similarly, different executions of *DO*-loops may be independent of each other, even though different code routes could be taken through the conditional statements contained in the loop.

In the case of the *Pipelined Vector* computers (e.g. *CRAY-1*, *CYBER 205*, etc.) all manufacturers have produced (mostly in *FORTRAN*) compilers that recognize when a *DO*-loop can be replaced by one or several vector instructions.

In the case that a lower (*Instruction*) level of parallelism is traced between *Phases* of instructions, then the processing of any instruction may be divided into several sub-operations using pipelining to overlap different sub-operations on different instructions.

Finally, at the lowest (*Arithmetic and Bit*) level of parallelism we have the option to define the *Arithmetic Logic* itself; namely, whether to proceed by performing the specific arithmetic in a bit-serial fashion,

or on all the bits of the number in parallel. Of course, intermediate possibilities exist, such as: To consider the number as subdivided into bytes†, then to proceed by taking the bytes of the number in a serial fashion, whereas the arithmetic logic has been performed simultaneously on all the bits of each byte of the number.

II.A.2.2: Explicit Parallelism Detection Approach

In the *explicit* approach to parallelism the programmer himself has to specify those tasks of a computational process which can be performed concurrently, by means of additional special instructions in the programming language itself. Although the addition of these parallel programming constructs is a time consuming and difficult to implement job, it has the significant advantage that the programmer is able to change the structure of the algorithm if it is not efficient for parallel processing.

Considerable research has been done on this approach and several concepts concerning the parallel tasks (namely, *declaration*, *activation*, *termination*, and especially *synchronization* and *communication*) have been considered.

In other terms, a *concurrent* program consists of sequential processes that are carried out simultaneously. These processes 'cooperate' on common tasks by exchanging data through shared variables. The problem is that unrestricted access to the shared variables can make the result of a concurrent program dependent on the relative speeds of its processes.

---

†*One byte is a sequence of 8 binary digits (bits).*

Dijkstra [*DIJK68*], in 1968, suggested the utilization of *semaphores* to prove many synchronization properties. Although *semaphores* have not reduced the requirements for large numbers of shared variables and consequently the interference potential is still quite large, they have been quite successfully utilized for a harmonious cooperation of a system of several processes.

After the semaphores, the *conditional critical region* concept was developed (see Hoare [*HOAR72*] and Hansen [*HANS73*]). The *conditional critical regions* assisted to reduce the potential interference by grouping the shared variables into resources, with an exclusive access to them and also allowing invariants on the shared variables established on a per region basis.

Later on, Campbell and Habermann [*CAMP74*], in 1974, suggested the utilization of *signals* and *path expressions* for synchronization purposes.

A further reduction in the processes interference led to the introduction of *monitors*[†] (see Hansen [*HANS77*]). This is a language construct that enables a programmer to tell a compiler how a shared data structure can be used by processes. In other terms, a *monitor* defines a shared data structure and all the operations processes can perform on it, synchronizing them and transmitting data amongst them; these synchronizing operations are called *monitor procedures*. It can also control the order in which competing processes utilize shared, physical resources.

Hoare (see [*HOAR78*,*HOAR80*], respectively), firstly, suggested that the parallel composition and communication of processes should be accepted as a primitive programming concept and, secondly, introduced a simple mathematical model for 'Communicating Sequential Processes' (*CSP*), proving

[†] *CONCURRENT PASCAL extends sequential PASCAL with 'concurrent processes', 'monitors','queues' and 'classes'.*

also the correctness of programs expressed as such.

The *CSP* concept formed the basis on which the *mutual exclusion* concept, in the case of *asynchronous* parallel processes, was developed (see Burns, et al [*BURN82*]). There is a vast difference between the sequential processing and the concurrent processing of several asynchronous parallel processes, since the execution of the latter depends on variables difficult to predict, such as the relative speeds of the processes, the operator's intervention, the interrupts, etc.

Without a doubt the most efficient way to prevent interference amongst the processes is by utilizing *critical sections*, which will include the section of code that every process attempts to execute at the same time; in other terms, the *critical section* can be executed by one at a time (i.e. in a sequential manner) process only. This mutual exclusion of accessing that section is ensured by means of *entry* and *exit* protocols to each critical section, protocols which also do a sort of *scheduling* thus determining which of the several contending processes is allowed to proceed each time. A more detailed study of this parallel construct can be found in paragraph (*II.A.3.1*).

For expressing concurrency, several mechanisms in the form of additional parallel constructs have been implemented; these will cope with the assignment of the available processors to the independent computations and also will allow the concurrently executing tasks to communicate, synchronize critical computations, etc., literally '*to look over each other's shoulders*'.

In addition, precautions must be taken to ensure the integrity of those areas in the main storage, shared by the various tasks. Each parallel path must be executed by only one processor, no matter how many

processors are 'attached' to the task, in order to protect the stability of the results; in other terms, when a processor is assigned to a path, all the others must be 'informed' of that and be locked out of this path.

Various forms of statements have been investigated; for example, *COBEGIN* (see Dijkstra [*DIJK68*]) or *PROCESS* declarations (see Hansen [*HANS75*]) identify the parts of a parallel program which can be executed in parallel, distinguishing the local variables from the shared variables. A programming language which allows process declarations, as well as, an arbitrary nesting between processes and procedures, is *ADA* - (see Ichbiah, et al [*ICHB79*]); this fact although it gives a great deal of programming power to the language, it makes very difficult to understand the effects of the so complex programs, since it leads to potentially more sharing and more complex execution paths.

Another example of the *explicit* approach is the *PARALLEL FOR* (see Gosden [*GOSD66*]), which takes advantage of parallel operations generated by the *for* statement in *ALGOL* and similar constructs in other languages. Also, the programming language *PL/1* provides the *TASK* option with the *CALL* statement, which indicates the concurrent execution of parallel tasks.

A different way to indicate the parallelism in the *explicit* approach is to write a language exploiting the parallelism in algorithms to be implemented by the operating system[†].

Anderson [*ANDE65*], in 1965, introduced the *FORK*, *JOIN*, *TERMINATE*, *OBTAIN* and *RELEASE* constructs for parallel processing; the form of these statements, in *ALGOL-68* format, is:

[*FORK* statement]::=*fork* [Label list];

[†]*An example of this case is the ALGOL-like TRANQUIL language which was utilized on ILLIAC IV.*

[*JOIN* statement]::=LABEL: *join* [Label list];

[*TERMINATE* statement]::=LABEL: *terminate* [Label list];

[*OBTAIN* statement]::= *obtain* [Variable list];

[*RELEASE* statement]::= *release* [Variable list];

where

[Variable list]::=[Variable],[Variable]/[Variable list],[Variable].

The [*FORK* statement] indicates the concurrent processability of a specified set of tasks within a process, initiating a separate control for each task under a different label. Only local labels must be used and their scope is defined as the block scope in which this statement is declared. The next sequence of tasks can be initiated as soon as all the emanated tasks, from a *FORK*, have reached a *JOIN* statement. However, in certain cases some of the parallel operations (e.g. a branch operation to alert an I/O unit for a momentary utilization), is not necessary to be completed for the processing to be continued. The release of these processors, without the initiation of further action, can be achieved by the execution of an *IDLE* statement (see Gosden [*GOSD66*]).

The [*JOIN* statement] is closely related with the previous statement, occurring in the same program level. This statement terminates the parallel paths that are involved in the *FORK* according to the Label list and a single path may follow. This action is implemented by compiling a code that causes test bits to be available, allowing the *FORKed* paths to be synchronized after they are completed. The *JOIN* statement label is the operand of the last *goto* statement appearing in each task generated by the *FORK* statement.

Alternatively, the *co-routines* concept, immediately emanating from the *FORK-JOIN* technique, allows independently executable routines to inter-

communicate during execution; in addition, they maintain some re-synchronization key-points to ensure that the computed values have been properly passed from one to the other. The *FORK-JOIN* technique is depicted in *Figure (II.A.2.2-f1)*.

The [*TERMINATE* statement] is used to explicitly terminate program paths (according to the included Label list), which have been dynamically activated by the *FORK* statement, thus avoiding the creation of a backlog of meaningless incomplete activations.

Actually, the *JOIN* and *TERMINATE* statements are *control counters*[†], decreasing by one after the execution of one statement, comparing each time to zero; if different than zero, the path is terminated and the processor is free to execute the next path in the queue, otherwise, the processor goes to the next program segment, exactly after the *JOIN* statement.

The last two powerful statements permit the *locking/unlocking* of variables, from access by other segments of the program.

The [*OBTAIN* statement] provides exclusive use of the variables in the Variable list. It is used to avoid mutual interference by *locking-out* other parallel program paths from the use of these variables. If this statement occurs in a block then these variables should be the same variables occurring in higher level blocks.

The [*RELEASE* statement] is the logical counterpart of the *OBTAIN* statement. It can be applied selectively since it only allows access (*releases*) to those variables (from the Variable list) that have been previously *locked-out* by an *OBTAIN* statement.

More specifically the *OBTAIN/RELEASE* concept is an approach

---

[†]*They are initialized to the number of labels in their Label lists.*

implemented to assist in solving the synchronization problem. However, these statements present many implementation difficulties since they occur just before the use of a variable. The execution of a data-fetch function determining (before performing the request) the status of the requested data or indirect addressing, can ensure the exclusive use of variables and arrays. The data request cannot be performed when, either the path executed by the processor awaits access (*suspended*) and the processor is reassigned to other work, or in the case of a processor being *dormant* by attempting to access *locked* data, until the data is released by a *RELEASE* statement.

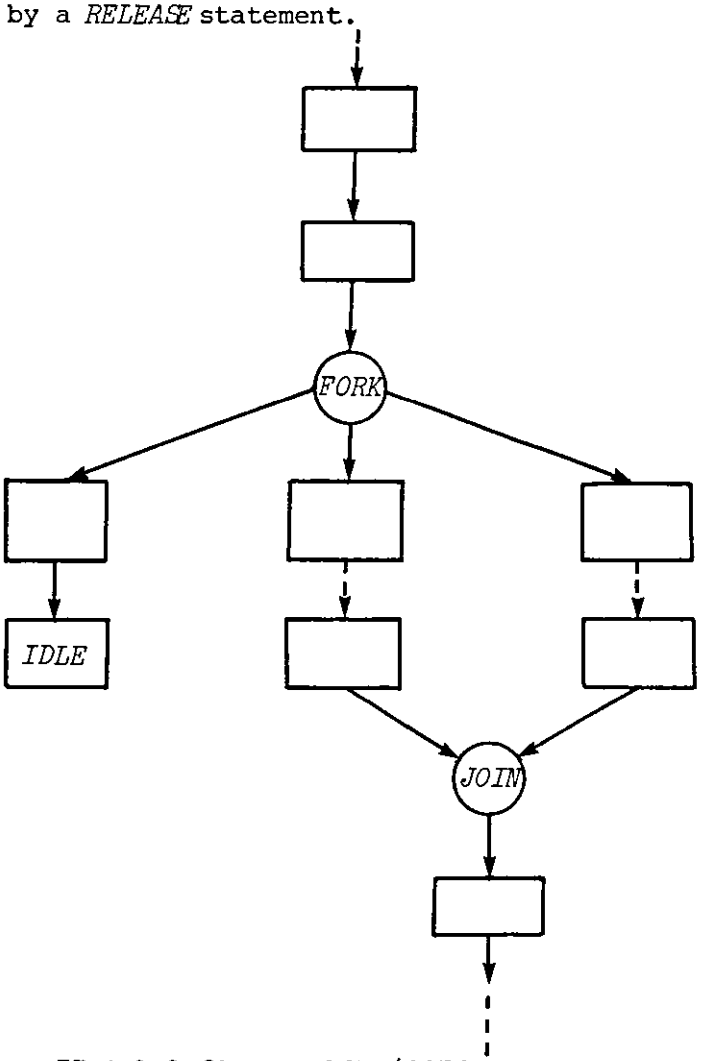


Figure II.A.2.2-f1: The *FORK/JOIN* Technique.

A concept similar to the *OBTAIN/RELEASE* is the *LOCK/UNLOCK* concept which was introduced by Dennis and Van Horn [*DENN66*], in 1966.

In conclusion, all the above mentioned statements are directed to the run-time Operating System and supply enough information to control parallel and multiprogramming activities. In particular, when a *FORK* statement is encountered, the compiler generates code to enter the run-time executive routine to create as many parallel paths as the number of labels in the Label list of the *FORK* statement. Each of these paths is assigned to the available processors and usually the first path is carried out by the same processor that carries out the *FORK* statement itself.

In the case that the number of processors is smaller than the number of paths then the excess paths are placed in a resources queue to wait for a *free* processor.

The labels contained in the Label list of the *FORK* statement are arranged on a special forward reference list, since they can be presented anywhere in the program. When a label is encountered, this list is searched and when the label is found it is removed from the list and a special heading information† is generated just before the labelled block, which may be fed in when the program segment is completely compiled.

To recapitulate, in the *explicit* approach the process of parallelization is under the entire responsibility of the programmer, a fact which jeopardizes program determinancy.

---

†*This information may include code length, data, etc.*

II.A.2.3: Implicit Parallelism Detection Approach

This approach to parallelism involves the *implicit* detection of parallel processable tasks, within programs intended for sequential execution. The determination of the *inherent* parallelism does not depend on the programmer, but relies instead on indicators existing within the program itself; in other terms, there is not such a need, as in the *explicit* approach, for the programmer to recode the sequential programs indicating the parallelism, for these properties can be detected using *implicit* recognition techniques. However, in contrast to the relative ease of implementation of the former approach, this is associated with sophisticated, complex compiling and supervisory programs.

A desired indication of the tasks is which of them can be executed in parallel and which must be completed before the next sequence of tasks commences; consequently, the detection process of the inherent parallelism can be distinguished into two parts - 'recognizing' the relationships between tasks within a level and 'utilizing' this information to indicate the *ordering* between tasks.

A variety of methods, some of which we subsequently present, have been proposed for an *automatic* recognition scheme to accomplish this detection; however, a *recognizer* which is universally applicable cannot be implemented since it is dependent on the source language.

As we have already mentioned, Bernstein [*BERN66*], in 1966, presented an inherent parallelism detection method in terms of sets representing memory locations, developing the deterministic conditions for the parallel execution of sequentially organized processes. His work is mainly based on four separate ways in which a memory location can be utilized by a sequence of instructions; in particular, these ways are:

i) The location is only *fetched* during the execution of τ_i task;

ii) the location is only *stored* during the execution of τ_i task;

iii) the first operation within a task involves a *fetch* with respect to a location.

One of the succeeding operations of τ_i task *stores* in this location; and,

iv) the first operation within a task involves a *store* with respect to a location.

One of the succeeding operations of τ_i task *fetches* this location.

However, although these conditions were sufficient to ensure the *commutativity* and *parallelism* of two program blocks, he showed the lack of proper algorithms for deciding these factors of arbitrary program blocks.

A complementary to Bernstein's work, was the work carried out by Fisher [*FISH67*], in 1967, which approaches the problem of parallel task detection in a general manner, formalizing in the form of an algorithm the above conditions. This algorithm utilized the input and output sets of each task (process) to determine the essential ordering and thus the inherent parallelism.

In 1969, Ramamoorthy and Gonzalez ([*RAMA69*], [*GONZ69*]), presented a new approach based on the oriented graph modelling of computational processes, developing also a *FORTRAN Parallel Task Recognizer*; in these graphs, the *vertices* (nodes) represented single tasks and the *oriented edges* (directed branches) represented the permissible transition to the next task in sequence. Consequently, the computational processes properties could be studied by simply manipulating a ($n \times n$) *Connectivity Matrix-C* (see Ramamoorthy [*RAMA66*]), utilized to express these graphs in computer

terms. The representing theory behind the matrix was that an element C_{ij} was a *1* if and only if there was a directed edge from node i to node j, or it was $\emptyset$ otherwise.

Another work carried out by Evans and Williams [*EVAN78*], in 1978, introduced a method of detecting parallelism in *ALGOL*-type programming and some particular language constructs, such as *loops*, *if* and *assignment* statements, were studied. For implementing these constructs, Williams [*WILL78*], in 1978, presented an *ALGOL 68-R* program describing how an existing multi-pass compiler can detect the potential parallelism. This compiler was extended by adding two more stages to it, the *Analyzer* and the *Detector* programs.

The role of the *Analyzer* was the limited subdivision of the main program into subprograms, the so-called *stanzas*; the size of a *stanza* would be a specific program construct (e.g. a *loop*) or a collection of statements utilizing fifteen different variables, the most.

The *receiver* of these *stanzas* was the *Detector* whose role was to determine the existing (if any) parallel relations between them. Subsequently with the information provided by the *Analyzer* and *Detector* programs would be possible the identification, during the compilation time of a sequential program, of those parts that might be executed concurrently.

A particular attention has been given for the detection of inherent parallelism within *arithmetic expressions*. It has been estimated that the time required to calculate an arithmetic expression on a conventional computer is proportional to the number of operations involved; however, the time required, for the calculation of the same arithmetic expression on a parallel computer, has been estimated to be proportional to the

number of levels in the tree representation of the expression.

The above can be apparently seen if we consider the following simple arithmetic expression:

$$A+B+C+D+E+F+G+H \quad . \qquad (II.A.2.3:1)$$

In *Figure (II.A.2.3-f1)* is depicted the representation of this expression for a serial and a parallel computer, respectively. It can be easily detected that this expression, sequentially estimated, requires *seven* units of time, whereas, in parallel, requires only *three* units of time (and a decreasing number of processors), as the number of representation tree levels, respectively.

A conclusion which may be drawn is that executing an arithmetic expression, in a parallel processing environment, the potential inherent parallelism is inversely proportional to the number of levels (or *height*) of the expression tree representation.

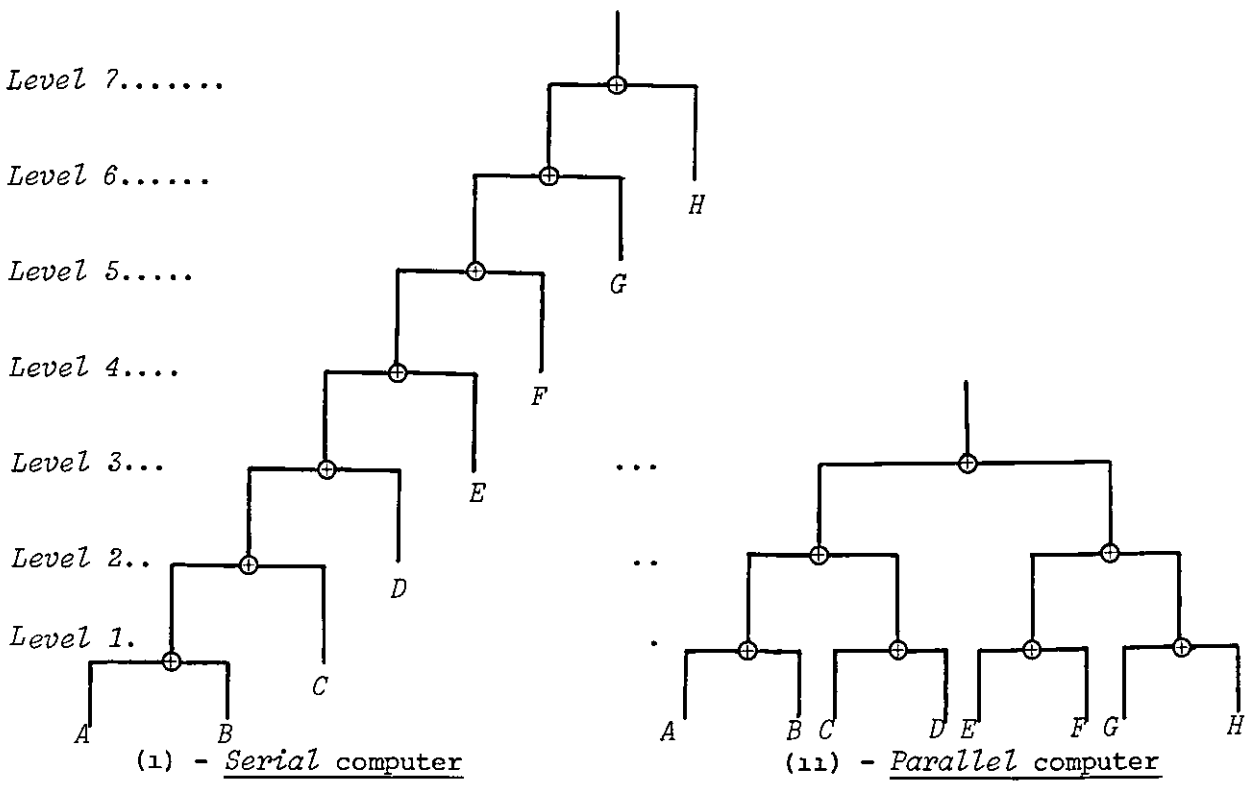


Figure II.A.2.3-f1: Binary Tree Representations of the Arithmetic Expression: *A+B+C+D+E+F+G+H.*

Many algorithms have been proposed for the detection of parallelism at the arithmetic expression level; some of them are those proposed by Squire [*SQUI63*], Hellerman [*HELL66*], Stone [*STON67*], Baer and Bover [*BAER68*], Ramamoorthy and Gonzalez [*RAMA69*], Kuck and Maruyama [*KUCK73*], Brent [*BREN74*], and Muller and Preparata [*MULL76*].

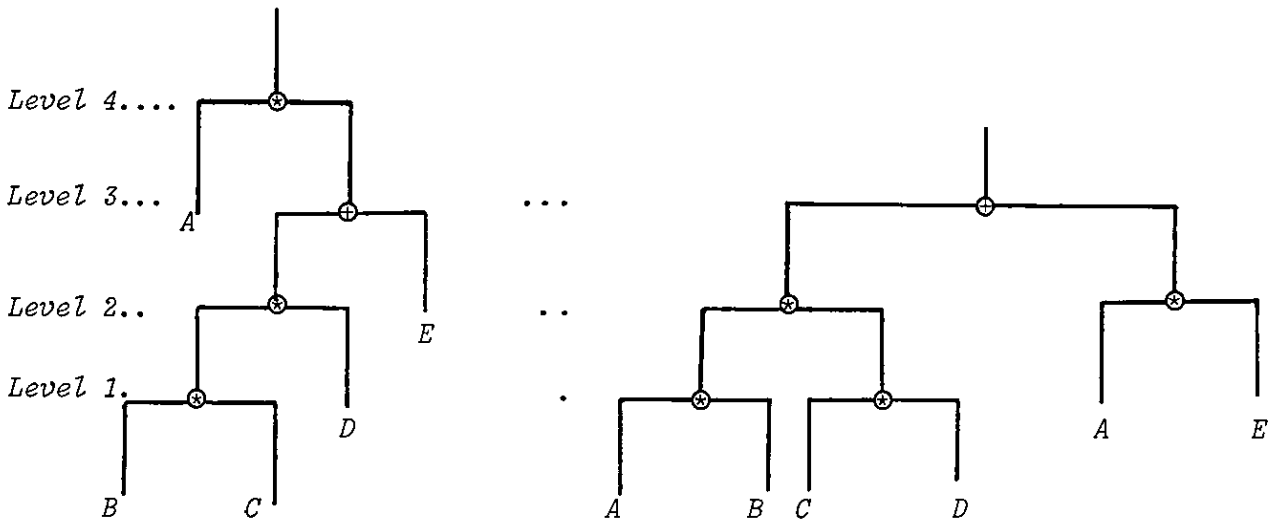
Although we have not attempted a complete survey of all the proposed algorithms, we should, in particular, mention the works carried out by Kuck [*KUCK77*] and Wang and Liu [*WANG80*].

Kuck examined the application of *distribution*† over arithmetic expressions, such that a tree representation is of minimum height. However, although this *distribution* form may involve some extra operations [see *Figure (II.A.2.3-f2)*] for the parallel execution, than for the sequential execution, the completion of the former execution is still faster, due to fewer operational levels.

Finally, Wang and Liu followed an innovative approach which did not concern with tree-height reduction techniques, as proposed by Squire, Stone, Baer and Bovet and Ramamoorthy and Gonzalez; they introduced the notion of the 'Parallel Execution String' (*PES*), which was utilized to detect parallelism not only at the arithmetic expression level, but also at the statement and the block levels. They, also,presented two algorithms to convert expressions into *PES's*, as well as the organization of a multiple microprocessor system designed for the parallel processing of them.

---

† *Since, by use of 'associativity' and 'commutativity', in some cases, no lower height tree representation could be found (e.g. the tree representation of the arithmetic expression in 'Figure (II.A.2.3-f2)').*



(i) - *Serial* computer

(ii) - *Parallel* computer (one more multiplication)

Figure II.A.2.3-f2: Tree-Height Reduction (by *Distributivity*) of the Arithmetic Expression: *A*(B*C*D+E)*.

II.A.3: PROGRAMMING CONCEPTS OF THE LOUGHBOROUGH MIMD MULTIPROCESSING ARCHITECTURES

The hardware and software characteristics of the *Interdata Dual Processor* and the *NEPTUNE(MIMD)* systems have been discussed in *Chapter I*. The programming concepts behind these systems will be presented in this paragraph.

The implementing programming language for both parallel systems is *FORTRAN*; especially for the latter system, which is an advanced extension of the former, since parallel programming facilities were required as quickly as possible, the choice of parallel language was limited to those available on the Texas 990/10. Between the *PASCAL* and *FORTRAN IV* programming languages provided, the latter was preferred, since, for example, it did not permit recursion and storage was statically allocated at compile time, i.e. there was no stack area to have to copy to other processors; consequently, its adaptation for parallel utilization was a far easier task.

This took place by means of several pseudo-*FORTRAN* syntactic constructs (i.e. *macros*), which were added to the language to achieve the parallel processing requirements. These constructs are converted to *FORTRAN* calls to machine code written routines by a preprocessor program running before the normal *FORTRAN* compiler.

In general, the user of the *NEPTUNE* system, similarly to the *Interdata Dual Processor* system, is required to define in his program and in a simple manner, the creation and termination of parallel paths, which data is shared between paths, and the synchronization to ensure the reliable update of certain shared data structures.

All paths that are created together must be terminated together;

in addition, each path must be executed by only one processor while at the same time the remaining processors must be informed and *locked-out* of that path, a fact ensuring the stability of the results.

From the data 'communication' point of view, an important programming construct which has been implemented on the *NEPTUNE* system, to maintain the utilization of variables which are shared amongst parallel paths, is the *$SHARED* construct. Data that is required by parallel paths and is not initialized in a *$DOALL* statement or a *FORTRAN DATA* statement, has to be defined as shared data using

$SHARED Variable List;

this forces the variables in the list to be loaded in the common memory, whilst the rest of the data, including the program code, is held in local memory. This construct has exactly the same properties as a normal *FORTRAN* labelled *COMMON* block, namely, it is a region of statically allocated storage sharable by different program units.

In any *MIMD* parallel processing system, 'synchronization' (or 'coordination') between parallel paths is required. In particular, when a path produces results that might be required by another path, synchronization is necessary to transform correctly these results, as well as, when parallel paths try to access shared resources; namely, these shared resources, which may be some shared data structures in the user program or a shared disk drive, must be accessed in a controlled manner to prevent them from being corrupted.

Various approaches have been proposed to enable accesses to shared resources to be synchronized. The algorithms proposed so far in the literature can be broadly classified into two groups, *resource master* and *bartering* (although some work has been couched in terms of communicating

sequential processes) - see Newman, et al [*NEWM84*].

In the case of *resource master*†, as the name implies, the resource is always 'owned' by one of the processors, with the ownership passing between the processors at the discretion of the current owner.

With *bartering* algorithms (see Lamport [*LAMP74*]) the resource is usually 'unowned'. When a processor wishes to access a resource it performs some bidding algorithm at the same time as any (and possibly many) other processors who also require the resource. This bidding algorithm ensures that the ownership is always held uniquely. Once the resource has been utilized, it is released, which then permits processors requiring the resource to restart bartering.

In the following, we shall discuss on a hierarchical development basis the implementation of the parallel programming constructs on Loughborough parallel processing systems.

The answer to the *lock-out* or *mutual interference* problem, mentioned previously, between the processors, was provided by the implementation of an *Abstract Resource Ring*‡. This ring consists of a set of abstract resources that are available to all processors in the system. A resource may be 'possessed' by only one processor at a time, which in sequence, after it finishes, will pass it to the next 'waiting-requiring it' processor.

The unique possessing of a resource, by only one processor at a time, can be ensured by transferring a resource on a 'giving', as opposed to a 'taking', basis. Four states of a processor are distinguishable:

(1) Not having, not wanting

---

†*A simply modified 'resource master' has been utilized on the 'Interdata Dual Processor' system.*

‡*First implemented on the 'Interdata Dual Processor' system.*

(ii) Having, not wanting

(iii) Wanting, not having

(iv) Wanting, having.

Due to the first two states and the fact that a resource is 'given' not 'taken', it is not possible to implement control passively on a flagging basis. However, a flag is set by each processor indicating whether it wants a particular resource or not, which can be seen by every other processor. The 'resource-want' message, from a processor, is sent, by means of an interrupt, around the system causing the other processors to try to give up their unwanted resources. The demanding processors cycle sending interrupts until they get the required resources.

Originally the system is initialized so that the resources are arbitrarily allocated to one processor. The ring structure is implemented in terms of two subroutines[†], the *GETRES(I)* and the *PUTRES(I)*, with resources *I=1,8;* the former one obtains exclusive use of the resource *I* and the latter one relinquishes 'ownership' of the resource *I*. In particular, they provide the exclusive use of a whole segment of a program to a processor, instead of the exclusive use of just the variables contained therein; in other terms, a program segment, intended to be exclusively utilized by a processor, is created into a resource *I* and is placed between these *FORTRAN* callable subroutines, namely:

CALL GETRES(I)
⋮
CALL PUTRES(I)
} Program Segment .

Later on, the direct *FORTRAN CALLs* to these subroutines were replaced by two preprocessor commands, (*$ENTER,$EXIT*), which are internally translated to the above subroutine *CALLs*, respectively.

[†]*Their role is similar to that of the 'OBTAIN/RELEASE' statements.*

Actually, to claim and release resources, this pair of constructs is implemented as follows:

$ENTER 'name 1'

(code)

$EXIT 'name 1',

which enforces sequential access to certain desired shared data structures to ensure their integrity. In other terms, a *critical section* of a program, i.e. one which requires single processor access, at a time, is embedded within this pair of constructs. The resources utilized must be declared with *FORTRAN*-like names, using *$REGION* 'list of names', the scope of this declaration being the next *$END* construct; also, the same resource can protect different critical sections in the program.

In addition, critical sections can include subroutine calls, as well as, claims for resources can be nested, but certainly under the risk of a possible *deadlock* situation, which was the entire user's[†] responsibility to prevent it. Moreover, the resources named on the region are bound to system entities at the preprocessor stage.

In particular for the *Interdata Dual Processor* system[‡] the *static* overhead involved in 'obtaining/releasing' a critical region is ~800μs.

On the other hand, the complexity of operations, performed on the structure within the critical region, defines the maximum *dynamic* or contention delay. Thus, in parallel path scheduling, for which operations on the list take ~700μs, the maximum dynamic delay is ~(800+700)μs, i.e. ~1500μs.

However, this is not the only dynamic delay associated with the implementation of critical regions. In fact, there is an additional

---

[†]*It is possible now to check parallel programs for 'deadlock' automatically and an experimental program has been written for this purpose.*

[‡]*The characteristics of the 'NEPTUNE' system will be discussed in a subsequent paragraph.*

dynamic delay when the required critical region is not immediately available to the *caller* processor; in this case, it demands from the other processors, via an interrupt line ring linking the processors, to relinquish their unwanted resources thus causing a 'servicing-interrupt' overhead of 125μs.

To continue our historical development retrospection, the *Interdata Dual Processor* system purely *sequential* layout of the actual program code, with nested loops separate counters and sequential indicators have to be set up for each level of nesting to ensure the *stable* and *correct* execution of the, therefore, extensive code, was causing inconvenience. This fact led to the introduction of some *macro*-commands in the code and a *dynamic scheduling list*.

These commands are expanded by an interpreter to the lengthy equivalent code, thus saving the programmer from a considerable amount of work; on the other hand, the list would include every path segment that had been referenced, which, together with sufficient information to ensure the paths correct execution, would be accessible to all processors.

In fact, the user has available *three* pairs of parallel constructs in order to *generate/terminate* parallel paths, the last one being additionally introduced to the *NEPTUNE* system. These are:

(1) $\$FORK\ L_1,L_2,\ldots,L_k;L_n$ (where L is a statement label)

L_1 *'code for path 1'*

GOTO L_n

L_2 *'code for path 2'*

GOTO L_n

⋮ ⋮

L_k *'code for path k'*

L_n *$JOIN ,*

which generates parallel paths with different code, forcing them, with the *GOTOs*, to terminate to the label appearing after the semicolon;

(ii) *$DOPAR(ALLEL)* L_1 *I=*I_1*,*I_2*,*I_3

'code'

L_1 *$PAR(ALLEL)END* ,

which generates and terminates $(I_2-I_1+1)/I_3$ unique[†] parallel paths, according to the values of the control variable *I* (i.e. I_1,I_1+I_3,...), with identical code.

On encountering the constructs, the parallel paths are created dynamically and consequently the I_1,I_2,I_3 values can be defined at program run-time.

We shall not extend more our reference to the *Interdata Dual Processor* system, since this system has been actually substituted by the newer, more advanced and with more processing units, *NEPTUNE* system; however, to conclude the *operating* picture of the first Loughborough parallel processor system, the user programs are developed, compiled and then written to disk on system *B*. The application program and the Resource Ring control routines (*GETRES*,*PUTRES*) are then loaded into the bottom (*private*) 32K bytes of memory attached to processor *B*. The Ring Data Structure and any user defined data appearing on *FORTRAN COMMON* statements are loaded into the top (*shared*) 32K bytes of *B's* memory. Finally, the contents of the private memory of *B* are then copied to the private memory of *A* and both systems are initialized (including the arbitrary allocation of the *Abstract Ring Resources*). Execution of the

---

[†]*This ensures that different paths evaluate different results.*

program then takes place in both processors.

The last pair of constructs is utilized to generate parallel paths with the same code, forcing each processor to execute the code once and once only; it occurs as:

```
        $DOALL L1
          'code'
L1      $PAR(ALLEL)END .
```

In actual fact, this construct is utilized to initialize the data, or to obtain the timing information.

In all the *three* pairs of constructs *nesting* is permitted (see *Appendix C-II*), but considerable care must be taken when handling the variables being utilized in the inner loops. Actually, before a parallel path is executed, all nested *'FORKs'*† must have their index variables set correctly, as these indices must be held in private memory; also, the local variables of a 'parent' path are not made available to the children. The flowchart in *Figure (II.A.3-f1)* illustrates the form that a program, for a *MIMD* computer, might have (see Bekakos [*BEKA81*]).

Two similar commands have been developed for the implementation of these constructs on the *NEPTUNE* system, the *XPFCL* and the *XPFCLD* (see *Appendix C-II*), the latter being more efficient than the other. As we mention in *Appendix C-II*, with the utilization of the *XPFCLD* command only a single descriptor block‡ is used for all the paths created by a *'FORK'*; in fact, the *$FORK* construct utilizes the same type of block as the *$DOPAR* construct. The transformation is as follows:

---

†*Either '$FORK' or '$DOPAR' or '$DOALL' construct.*

‡*Blocks containing information about the paths (otherwise called 'Task Control Blocks' - 'TCBs').*

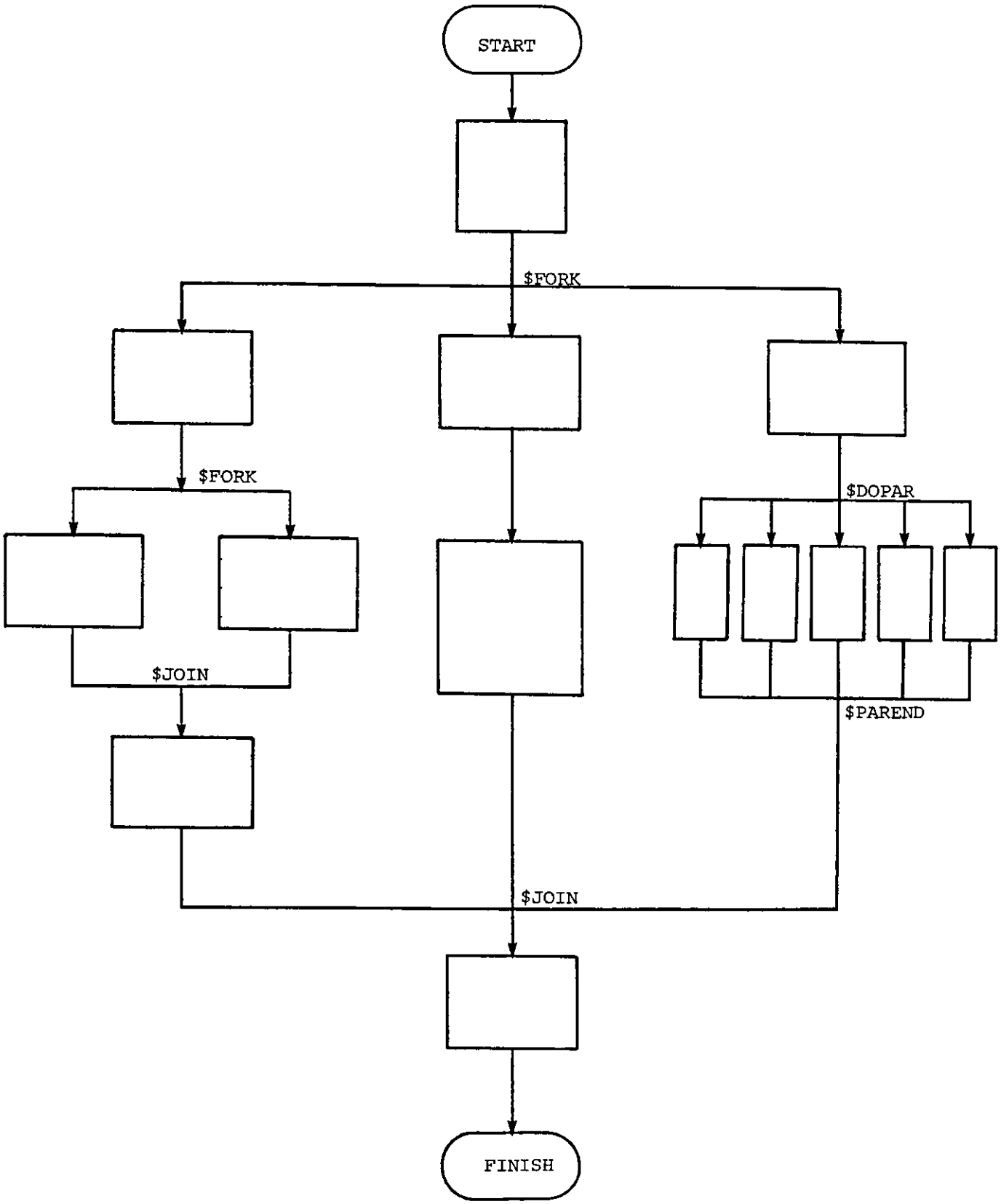


Figure II.A.3-f1: The Flowchart Structure of a Program for a *MIMD* Computer.

$FORK $L_1,L_2,L_3,\ldots,L_k;L_n$ (where L is a statement label)

to

$DOPAR L_n $0\emptyset01=1,k$

GOTO $(L_1,L_2,L_3,\ldots,L_k),0\emptyset01$ (FORTRAN computed GOTO)

$\vdots$

L_n $PAREND .

In addition, the processor that executes the 'FORK' construct also takes the first path from the scheduling list to execute it, as well as, it executes the path following the 'JOIN'† construct, after all paths from the $FORK or $DOPAR have terminated; this secures the variables track and maintains their correct setting, since the variables not utilized in the parallel paths are stored in private memory. In other terms, those private variables utilized in the path preceding the 'FORK' construct and required again in the following the 'JOIN' construct path, but not utilized within the 'FORK' created paths, will have the correct values only in the program-copy, stored in the private memory, of the processor that executed the path preceding the 'FORK' construct. Consequently, since the TCB, for the path following the 'JOIN' construct, is created when the 'FORK' construct is encountered, it is necessary that the number of that processor to be included in the TCB in order to enforce its reselection.

In conclusion, some other overall necessary constructs, which are essential in any parallel program to be implemented on the NEPTUNE system, are the $USEPAR, $STOP and $END constructs.

The last two constructs simply replace the normal STOP and END statements of FORTRAN, respectively; the latter one forces checking,

---

†Either '$JOIN' or '$PAREND' construct.

at pre-compile time, that the nesting of parallel syntactical constructs is complete within each individual subroutine, whilst the former one ensures the graceful program termination.

Finally, the *$USEPAR* construct must be the first parallel statement to be executed; on encountering this construct, all but one processor are forced to 'wait' until parallel paths are created for them to execute.

II.A.3.1: THE USER INTERFACE TO THE *'NEPTUNE'* PARALLEL PROCESSOR SYSTEM

In the previous *Chapter* (*par.-I.B.5.3.2*), it was mentioned that the 'System Command Interpreter' - *SCI* is the user interface to the parallel system by providing several ways in which commands may be issued, e.g. at the simplest level, a sequence of *Menus* drives the potential user to the final desired list of commands.

In this paragraph, we present briefly and in a hierarchical sequence those standard particular commands which have been introduced to simplify the user interface to the parallel system and are, on a common basis, utilized in order to implement any program on it.

To *'log'* onto the system, having connected via a modem, type the sequence *'ESCAPE'*, *'exclamation mark'* (for the *TTY/820* hard copy terminals), or *'blank'* (red key), *'exclamation mark'* (for the memory mapped *911 VDTs*). This forces the *SCI* to respond with its title, requesting the user's *'IDENTIFICATION'* and *'PASSWORD'* (echoed as spaces); if the provided information is 'valid', then the *SCI* prompts back '[ ]'†.

The next task involved is the *'editing'* of the specific program to create a 'new' file into a directory (see *par.-I.B.5.3.2*). The editor is invoked by typing directly, i.e. the *Menu* sequence is skipped, the command

† *Indicates that commands may be typed in.*

'*XE*'- e*X*ecute *E*ditor; the *SCI* responds:

'*<INITIATE TEXT EDITOR>*'

'*<FILE ACCESS NAME:>*',

and since a 'new' file is under creation, by pressing the '*<RETURN>*' key it displays an '*<*EOF>*' - *E*nd *O*f *F*ile prompt on top of the screen.

By pressing the '*<F7>*' button once† a new line is fed in automatically every time the '*<RETURN>*' key is pressed; note that only 'upper case' letters are allowed for the actual program code.

As soon as the typing of the program is completed, the '*QE*'- *Q*uit *E*ditor command is invoked forcing the *SCI* to respond:

'*<QUIT EDIT>*' *(first page)*

'*<ABORT?:NO (by default)>*';

by answering the '*ABORT?*' with '*YES*' the edit phase is to be lost, whilst by pressing the '*<RETURN>*' key a new screen will display:

'*<QUIT EDIT>*'

'*<OUTPUT FILE ACCESS NAME:>*' *(second page)*

'*<REPLACE?:NO (by default)>*'

'*<MOD LIST ACCESS NAME:>*' .

The first question is filled in with the name the file is to be known by; a response to '*REPLACE?*' with '*YES*' will overwrite an already existing file, whilst the last question allows one to put 'the modifications' made into a file named at this point. If no name is given the modifications are not saved.

Sometimes a '*QE*' command may 'fail', but the user remains in the editor until a subsequent successful one, without losing the work carried out in the edit phase; the most common reasons for failure are,

---

†*A second press brings us back to the manual line advancement mode.*

either as the answer to the *'REPLACE?'* question has been considered the system default one, while already a file exists under this name, or when the 'file protection' has not been erased, or there is not space in that directory (or even directory), or finally, another user's program, running in the background, has the file open.

After the program has been fed in the system the next phase can be started, involving the *'Compiling/Linking'* of the created program, in an attempt to produce the so-called *'Load Module'*. There are several commands related to the creation, deletion and installation of load modules. The most commonly (up to now) utilized command, to create load modules from the user's source program, is the *'XPFCL'*† - eXecute Parallel Fortran Compile and Link command (see *Appendix C-II/par.-II.A.3-i*). The effect of this command is to:

i) Preprocess the user's source program to translate the utilized parallel constructs into normal *FORTRAN* statements,

ii) compile the resultant *FORTRAN* code,

iii) link the compiler output with the available *FORTRAN* libraries and machine code written routines to control the parallelism, and

iv) store the created load module in the user's program file.

A progress report message, in the form of *'ERRORS/NO ERRORS'*, is available for each of the above stages; certainly, the procedure of the command will be interrupted if an error is detected in one of these stages and an additional, to the progress report, more explanatory message will be written to that stage file, which can be viewed using

†*Certainly, (see Appendix C-II/par.-II.A.3.-ii), there are now available the 'XPFCLD/XPFCLX' commands which are more powerful than the 'XPFCL' command.*

the '*SF*'-Show File command.

In *Figure (II.A.3.1-f1)* we can see the actual terminal display of the *XPFCL* command, where '*<SOURCE:>*' asks for the name of the source program file,

'*<FMPLIST:>*'for a filename for the output of the preprocessor stage,

'*<COMLIST:>*'for a filename for the compiler listing,

'*<LINKLIST:>*'for a filename for the linker listing, and

'*<NAME:>*'for a name by which the load module will be known thereafter.

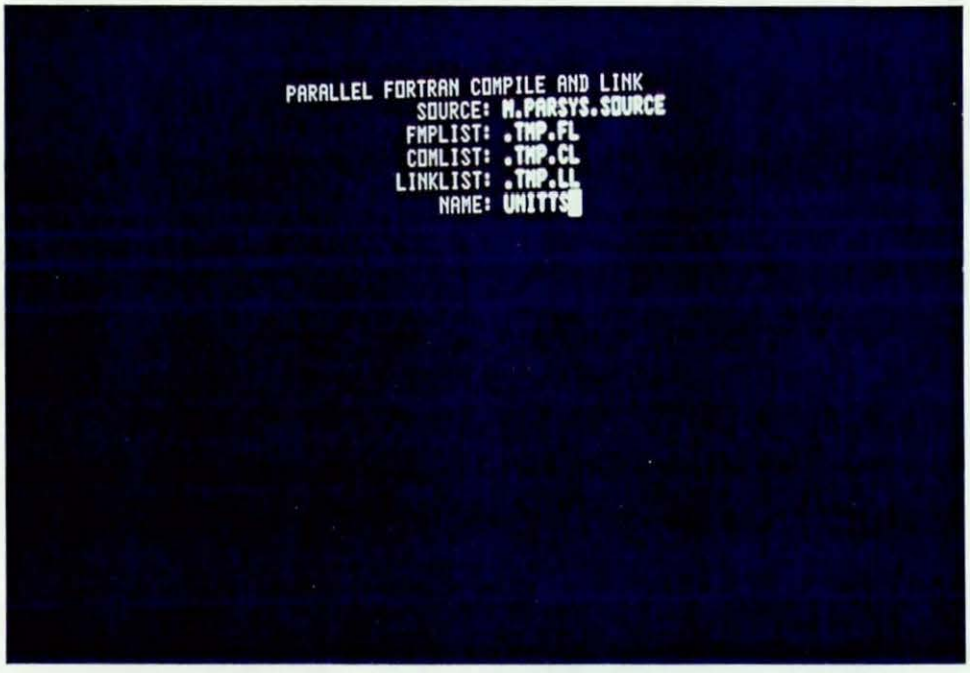


Figure II.A.3.1-f1: The *XPFCL* Command.

Some other commands related to the *Compiling/Linking* command are, the '*DPT*'-Delete Parallel Task, the '*IPT*'-Install Parallel Task, the '*MPF*'-Map Program File, the '*WAIT*'-*wait* for background task, and the '*KPT*'-Kill Parallel Task commands.

For the first two commands, the system responses: '*<NAME:>*',for the

name of the load module. The *DPT* command allows the user to delete a named load module, before installing a new copy. This deletion procedure is necessary, because it is not possible to overwrite an existing load module; however, each user is allowed up to *256* differently named load modules.

In the case that the install phase of the *Compiling/Linking* command 'fails', because the installed name already existed, the program can still be installed from the 'linked' output, by utilizing the *IPT* command, thus saving the re-issue of the *Compiling/Linking* command. The *DPT* command will delete the old version of the program, prior to using the *IPT* command, or the *IPT* command can be utilized to install the program under a different name.

For the *MPF* command the system responses:

'*<LISTING ACCESS NAME:>*',

listing the names of all user's load modules, for the case the required load module name has been forgotten. The user may specify a file for the listing, however, the default response, by pressing '*<RETURN>*' without any file or device name given, is the user terminal.

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

To 'kill-off' a parallel task running in the foreground, the user must press the '*<ESCAPE>*' key, followed by the '*<CTRL>*'† and '*<X>*' keys

†*The abbreviation for CONTROL.*

(for the *TTY/820* hard copy terminals), or the '*<Blank>*' (Red Key) followed by the '*<CMD>*'† key (for the memory mapped *911 VDTs*).

After having concluded the above phase successfully, the user enters the last and most important phase, that of the actual '*running*' of the load module.

An essential task, which must be carried out before the running of the load module, is the determination of the *I*nput/*O*utput channels utilized in the *FORTRAN* program; by utilizing the command '*AS*' - *A*ssign *S*ynonym value‡ the system responses:

'*<SYNONYM:>*'

'*<VALUE:>*',

where the '*SYNONYM*' waits to be assigned with an I/O channel* each time, whilst the '*VALUE*' waits to be assigned with a file or device name. Alternatively, the value '*ME*' can be given, implying a direct I/O communication with the user terminal.

Parallel programs are initiated from one processor, the processor that the user is 'logged in' on; the system then starts up copies of the program on the other processors that have been requested.

To run a parallel program the '*XPFT*' - e*X*ecute *P*arallel *F*ortran *T*ask command is utilized. The user declares (see *Figure (II.A.3.1-f2)*) on which processors the specific program should run, the name of the load module and finally whether this execution is required in the foreground or in the background (multi-tasking) management environment. However, background tasks should not involve I/O with the terminal and commands (the majority of the utility commands for the *NEPTUNE* system can be

---

†*The abbreviation for 'COMMAND'.*

‡*This command can be utilized to set or clear synonyms.*

**All 'FORTRAN' I/O channels are characterized by the word 'UNITm', where 'm' is the channel number.*

found in the [*Texas Instruments, II & IV*]) are available to inspect the background status, since the *SCI* is still running thus being available to process user requests.

The processors in the system are numbered $\emptyset$ to *3* and any combination of them can be given, as long as the initiating processor (the one 'logged in' on) is included in the list. 'Errors' occurring at run-time and terminating conditions, are reported to the user, whereas a listing of the most common ones is given in [*Texas Instruments, VI*]. A correct execution produces the report

'*<STOP $\emptyset$>*'

'*<NORMAL PROGRAM COMPLETION>*' ,

for every executing processor (an actual example of a'non-running' parallel program can be seen in *Figure (II.A.3.1-f3)*).

In addition, a more advanced execution command has been developed, the '*XPFR*'- Repeated *XPFTs*, which remembers the previously given list of processors, as well as, it displays the output unit automatically at the end.

Finally, there is a most important command, the '*SOPR*'- Set up Overnight Parallel Run, which allows timing runs to be made overnight, rather than requiring exclusive use during the day. A user, in one *login* session, can request up to *ten* overnight runs, each consisting of a number of executions of the same program, with the same input data on various processor combinations, terminated by pressing the '*<RETURN>*' key in response to the prompt. The listings from the runs are concatenated into the file designated by the *SOPR* 'output file' prompt, which will be on the users' temporary[†] file Directory used for compilation listings.

[†]*The users' source disk 'PARUSER1' is 'write' protected during the night.*

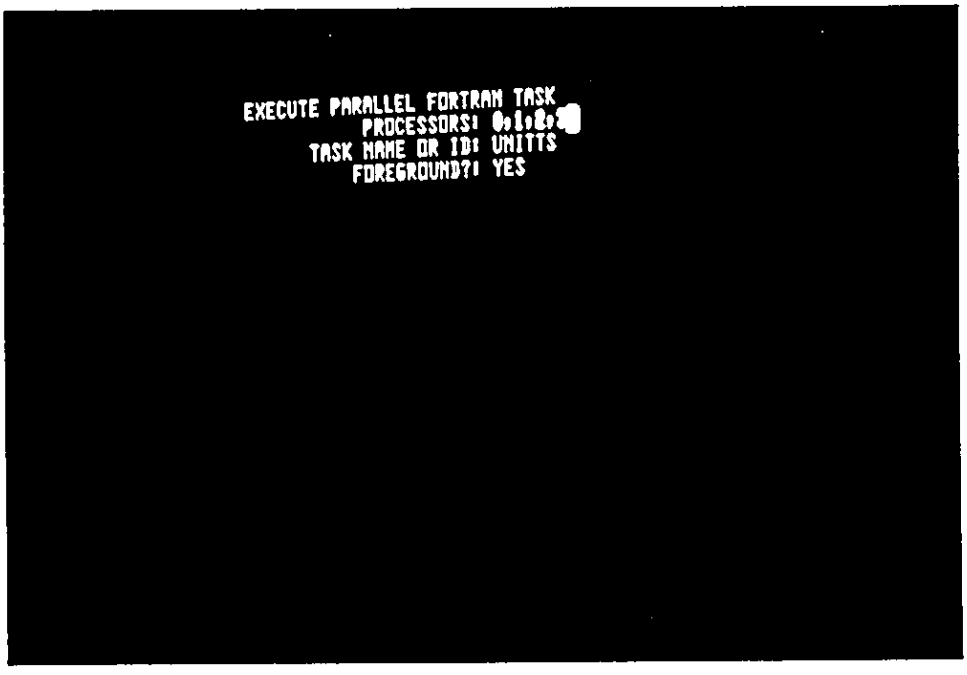


Figure II.A.3.1-f2: The *XPFT* Command.

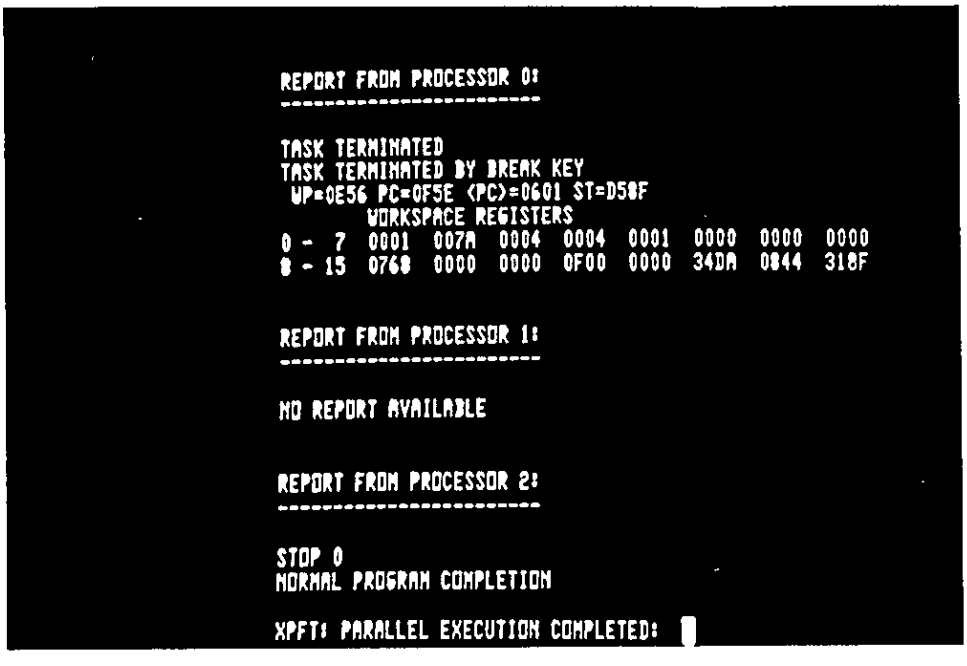


Figure II.A.3.1-f3: The Report of a *'Non-Running'* Parallel Program.

The command 'XOBD' - Delete XOB† queue entry, can delete the number of a wrong SOPR entry, which entry number can be found using the 'XOBQ' -XOB Queue listing command.

To conclude, as the overall throughput of SISD computers improves by multi-tasking, in the same way a MIMD computer can improve the utilization of processors by two different multi-tasking methods:

i) By using different processors to run sequentially each, independent programs in parallel, and

ii) by using disjoint sets of processors to run several parallel programs.

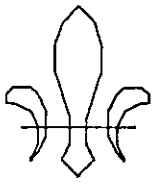
The inherent parallelism, for the majority of generalized problems, lies between 30 to 100 in potential speed-up, consequently, the utilization of a large MIMD computer of approximately 1000 processing elements, to run a mix of programs, could produce an overall throughput speed-up of perhaps 200-500.

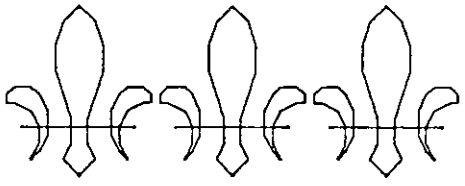
---

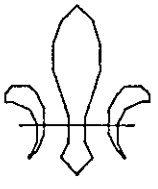
†'XOB' - eXecute Overnight Batch.

SECTION B

Aspects Of Parallel Algorithms Design And Analysis Methodology







'Nil Posse Creari De Nilo.'

'Nothing can be created out of nothing.'

De Rerum Natura, i.155

Of Epicurus

Lucretius 94?–55 B.C.

II.B.1: Construction Principles For Efficient Parallel Algorithms

Def.: A procedure consisting of a finite set of unambiguous rules, which specify a finite sequence of operations that provides the solution to a problem, or to a specific class of problems, is called an 'algorithm'.

Despite the continuing dramatic decline in the cost of hardware, which in coming years will make it feasible to economically build computers with a hundred thousand, or even a million processing elements, and to the contrary of the chronological evolution, the development of a parallel computer should only be considered as a second stage towards the algorithmic solution of a specific problem. However, although, and this is fundamental, the solution relies on the concept of the parallel algorithm more, than on the concept of the parallel computer, since the complexity of a parallel algorithm depends very much on the parallel architecture on which it is due to run, it is necessary to keep the architecture in mind when designing the algorithm.

Certainly there exists, at least, a substantial amount of parallelness in a surprising variety of problems, but the greatest constraint of all lies in us, in the form of the 'serial way of thinking'; in other terms, as we learn to formulate parallel algorithms and write parallel programs, we should also learn to 'think', at the *'conscious'*, as well as, the *'unconscious'* level, in an increasing parallel mode, there is a crucial need to develop our powers of parallel thinking.

This is the main reason that researchers should get involved with the system network and its structure, considering together problem, program, network structure, mapping of program onto that structure, and flow of data and other kinds of information through that structure. It is apparent that being fully aware of the system structure can prove itself to be a powerful means for developing more efficient algorithms.

Today, a wide range of parallel algorithms have been explored. For example, Kuck, et al [*KUCK77a*], Kuck [*KUCK78*], and Kung [*KUNG80*] have investigated numerical algorithms; Hanson and Riseman [*HANS78*], Tanimoto and Klinger [*TANI80*], and Preston and Uhr [*PRES81*] have dealt with perception programs, whilst Boral [*BORA81*] and Fishburn [*FISH81*] with database and artificial intelligence problems.

To actually construct a parallel algorithm there are two basic concepts: The historically *older* and weaker[†] concept is that of indirectly transforming the sequential algorithm into a parallel one, to be adapted to the given parallel computer architecture; the *newer* concept starts more naturally with the problem itself and investigates its potential inherent parallelism.

[†] *For efficient sequential algorithms do not necessarily lead to efficient parallel algorithms; to the contrary, sometimes inefficient sequential algorithms may lead to efficient parallel algorithms.*

Both of these concepts are relatively acceptable depending on the case-in-hand. In almost all cases we have a problem that we wish to attack. Occasionally, we can pose that problem precisely and definitely, and develop a solution procedure that can be formulated and coded for a computer. In such a case, the situation is rather a clear-cut, since it becomes a simple matter to define the inherently contained independent computations, thus developing more parallel but entirely equivalent algorithms.

As an example of such independent computations, consider the addition of two n-vectors $\vec{a}$ and $\vec{b}$, to yield another vector $\vec{c}$, i.e.

$$\vec{c} = \vec{a} + \vec{b} \ , \qquad \textit{(II.B.1:1)}$$

where,

$$\vec{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \ , \quad \vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \ , \quad \text{and} \quad \vec{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \ . \qquad \textit{(II.B.1:2)}$$

Apparently, the evaluation of the components of the result vector $\vec{c}$ is of the form,

$$c_i = a_i + b_i, \quad \text{for } i=1,2,\ldots,n \ , \qquad \textit{(II.B.1:3)}$$

and so the calculations are independent. Consequently, a computer with n processors can compute the result in one time-step, by evaluating the formula *(II.B.1:3)*, for each value of i, on a different processor.

However, much more often we come across a fuzzy problem, really a whole set of problems, or a whole realm of obscure and unknown issues, about which we would like to know more. For example, we wish to better understand and forecast the weather, we want perceptual systems for robots to recognize, grasp and manipulate objects; we develop programs to hack away at these problems, to give partial solutions or simply new

information and greater insight, programs that exhibit little or no inherent parallelism. In such cases, we should go back to the problem set, to our realm of inquiry, to start afresh in developing more parallel approaches to the problem. Certainly, sometimes we shall succeed only at the price of losing some of the information provided by the original algorithm; but, more often, we shall develop a powerful parallel algorithm.

The notion of parallel algorithms, and this is not surprising, has often been confused with the notion of programming an actual parallel computer; this had led to the classification of parallel algorithms according to the parallel architecture best suited for them (e.g. *SIMD* algorithms, *MIMD* algorithms, etc.). We think it would be better to follow the typology first proposed by Kung [*KUNG76*,*KUNG80*], since it is not only based upon parallel architectures, but also, upon the inter-communication features required amongst the various processing modules; according to this typology all algorithms can be classified into two broad categories, the *Synchronized* algorithms category and the *Asynchronous* algorithms category.

Because of historical and technical reasons (namely the existence of the *ILLIAV IV* computer), the majority of the literature, since the early sixties, has referred to the first category of the roughly mentioned as 'parallel algorithms for *SIMD* systems' (see the surveys by Miranker [*MIRA71*], Sameh [*SAME77*], and Heller [*HELL78*]); however, the relatively recent and phenomenal advances in technology have given to the second and scarcely investigated, up to then, category of *Asynchronous* or roughly 'parallel algorithms for *MIMD* systems' a considerable 'push' towards re-establishing a certain balance with the first category.

Despite the apparent technological construction problems, constraining the physical expansion of *MIMD* systems, compared to *SIMD* ones, researchers are still interested in investigating and comparing these two different types of systems, each with its advantages and disadvantages, to exploit their differences in parallelism.

As was exemplified by the various architectures of existing parallel computers, parallelism can be achieved in a variety of ways; the same holds with parallel algorithms, since a close correspondence must exist between architectures and algorithms. Consequently, since there exists such a variety, the question to answer now is, 'How to choose amongst the different alternatives in order to solve a specific problem, or what type of problem is better adapted to a given architecture?'.

Since, in most cases, *Performance* is the reason why parallelism is being investigated, it must be considered as a very critical issue. The study of how to design algorithms, for different parallel architectures, might reveal that an algorithm requires a peculiar feature of that architecture to run efficiently.

The *Performance* of an algorithm is defined by the absolute arithmetic answer to some relevantly set quantities, such as, the *Computation time*, *Speed-up* and *Efficiency* of the algorithm.

The actual *Computation times* are often proportional to the total number of arithmetic operations in the programs, whilst, in cases of programs with little arithmetic, are proportional to the number of memory accesses, or the number of I/O transactions.

In general terms, similarly to what was discussed in *(Appendix C-I/par.-I.B.4.1)*, the *Speed-up* ratio S_{nop} [†] is defined as:

[†] $S_{\underline{no.}\ of\ \underline{p}rocessors}$.

$$S_{nop} = \frac{\textit{Computation time on a serial computer}}{\textit{Computation time on the parallel computer}} = \frac{T_s}{T_{nop}} \; ; \qquad (II.B.1{:}4)$$

and, the *Efficiency* ratio E_{nop} is defined as:

$$E_{nop} = \frac{\textit{Speed-up ratio}}{\textit{No. of processors}} = \frac{S_{nop}}{nop} \; . \qquad (II.B.1{:}5)$$

For a parallel organization that can support n simultaneous processes, the ideal *Speed-up* and *Efficiency* ratios are equal to n and 1, respectively, but these are seldom (or never achievable). In order to achieve a 'fair' comparison, we must, always, compare the *best* sequential algorithm, to the *optimum* parallel algorithm, even when the two algorithms are quite different.

Computations that are very efficient to parallel computer systems, have a Speed-up ratio of $k.n$ (i.e. linear to n), where k is a constant near unity, but strictly less than unity; such a result is attainable in problems that have a natural iterative structure. In some cases, problems have Speed-up ratios of $O^{\dagger}(k.n/log_2 n)$, but these results are less desirable, although still well acceptable, since the speed can be improved by doubling the number of processors.

The dividing line between problems well-suited and poorly-suited for parallelism, lies between the Speed-up ratios of $O(k.n/log_2 n)$ and $O(k.log_2 n)$. An algorithm with a Speed-up ratio of $O(log_2 n)$, exhibits very little speed increase when we double the number of processors, so that such problems are usually best suited to serial computers, or to computers with a very limited amount of parallelism. In *Table (II.B.1-t1)* we present some typical problems which have been solved on *SIMD* computers,

---

†*Order*.

as well as, their Speed-up ratios.

To conclude, despite the fact that, since human time becomes increasingly valuable and costly and computers become cheaper to build, parallel computers are so indicated, they are far from easy to program. This can only become true after we have developed 'proper' languages and interactive program-development systems, to invite, suggest, encourage and even gently 'push' the researcher to be parallel, by helping him to explore and pin down the possibilities of parallelism. Therefore, all parallel programmers should be given as much help as possible, and applications programmers should be burdened as little as necessary, since, as the up-to-date situation stands, the researcher/programmer has to do as much of the work as possible, in order to give the system as much help as he can.

For the future, as many of us 'think in our own language', we should, some day, reach out the level where we shall 'formulate in silicon'; that is, we shall move from problem, to an anatomy-physiology formulation, that embodies our solution procedure into an architecture complex, through which an algorithm flows information.

In the remainder of this *Chapter*, we shall examine some particular schemes and techniques utilized to design algorithms, to efficiently exploit the parallel hardware potential of the different categories of computer systems. Finally, specific emphasis will be given to the particular parallel features and performance analysis characteristics of the *NEPTUNE* parallel processor system, at Loughborough University, since one of the goals of this Thesis is the design and analysis of certain parallel algorithms to exploit the *NEPTUNE* parallel processor system potentiality.

Algorithms	*Speed-up ratio*
Matrix computations and mesh calculations (Kuck [*KUCK68*])	$k.n$
Sorting (Stone [*STON71*]), tridiagonal linear systems (Stone [*STON73a*]) Linear recurrence relations (Kogge and Stone [*KOGG72*]) Polynomial evaluation (Munro and Paterson [*MUNR71*])	$k.n/log_2 n$
Searching (Karp and Miranker [*KARP68*])	$k.log_2 n$
Certain nonlinear recurrence relations, certain compiler processes	k (independent of n)

Table II.B.1-t1: Typical Problems Solved on *SIMD* Computers.

II.B.2: Schemes And Techniques To Design Algorithms To Map Onto Simd And Pipelined Vector Computer Architectures

We shall consider the design of parallel algorithms for these two categories of parallel computer architectures together, for, as has been mentioned earlier in *Chapter I*, the same essential approach is followed, since the same algorithms can be very efficient for both types of systems.

In general, we can say that algorithms executed on a *SIMD* computer require a high degree of parallelism, because this type of system possesses up to $O(n^m)$, $m=2,3,4,\ldots$ processors. The processors of *SIMD* computers are *synchronous* and consequently, they cannot be utilized to execute independent, but not identical, computations; in addition, they must remain *idle* when not required.

One of the most significant factors, influencing the system performance, is the 'intercommunication' required amongst the various processing elements, tasks or processes; consequently, algorithms should require a very 'regular' (and thus inexpensive) intercommunication between the modules (in this respect, they resemble *VLSI* algorithms, which will be discussed in *Chapter III*), following a pattern coinciding with the existing processing elements network.

In particular, non-identical computations will be executed sequentially on a *SIMD* computer, a fact which reduces the overall system performance.

A first partial conclusion which can be drawn up to here is that to design efficient algorithms for *SIMD* computers, one should only consider problems with substantial inherent amounts of identical independent computations; therefore, this means that all matrix and vector operations are well-suited to *SIMD* computers, but of course in each particular case

one must consider the corresponding number of processors, for the result to be obtained in exactly one step each time.

In a similar way, algorithms designed for *Pipelined* type of computers should be presented as a long string (the longer the string the greater the efficiency achieved) of identical operations that will be treated in an assembly line fashion.

A simple example, ideally applying the notion behind *SIMD* computers, is the *sum* of the two n-vectors given in *(II.B.1:2)*, where the n independent computations can be assigned to n different processors to be all carried out in one time-step, simultaneously. Certainly, this sum can be generalized to the addition of two $(n \times m)$ matrices A and B, where every row of A is added to every row of B and the required addition is as defined in formula *(II.B.1:3)*; obviously, the addition can be performed in one time-step by utilizing $(n \times m)$ processors.

In a similar way we may consider the matrix product:

$$C = A.B \ , \qquad \textit{(II.B.2:1)}$$

where A,B are $(n \times p),(p \times m)$ matrices, respectively, defined as:

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

$[b_{ij}]$, $i=1,2,\ldots,p/j=1,2,\ldots,m$; the result matrix C is defined as:

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} \ , \text{ for } i=1,2,\ldots,n/j=1,2,\ldots,m. \qquad \textit{(II.B.2:2)}$$

Again the number of independent computations is $(n \times m)$ and so the result can be obtained in one time-step by utilizing $(n \times m)$ processors.

The execution of arithmetic operations is also one of the most successful applications of *Pipelined Vector* computers, especially when the same sequence of operations is invoked very frequently, so that the

start-up[†] time becomes, relatively, insignificant. In particular for *vector programming*, it appears as a challenge to the software specialist. Areas where advances are specifically required include the following interrelated topics:

(i) Algorithmics (algorithms for vector processing, and methods for finding such algorithms);

(ii) program design (how to find program and data structures which will lead to efficient use of supercomputers, while ensuring other program qualities, such as, reliability, clarity, portability, modularity, etc.);

(iii) program transformation (methods for adapting existing programs to efficient execution on vector computers);

(iv) languages for vector programming; and,

(v) proof methods.

Pipelined algorithms, for floating-point additions, multiplications, divisions and square roots, have been discussed in Chen [*CHEN75*] and Ramamoorthy and Li [*RAMA77*], where the connection amongst the various stations of the pipe is linear.

As an example, let us consider a *k-pipelined digit adder* (see *Figure (II.B.2-f1)*), the description of which is given by Chen [*CHEN75*]. Assume that two n-vectors, $\vec{U}=(U_1,U_2,\ldots,U_n)$ and $\vec{V}=(V_1,V_2,\ldots,V_n)$ are to be added. Let $U_i=u_{i1},u_{i2},\ldots,u_{ik}$ and $V_i=v_{i1},v_{i2},\ldots,v_{ik}$, to be the binary representations of U_i and V_i, respectively. The u_{ij} and v_{ij} approach the linear array of modules in a synchronous fashion mode. At each cycle, each module *sums* the three numbers arriving through the corresponding

[†]*The initialization time.*

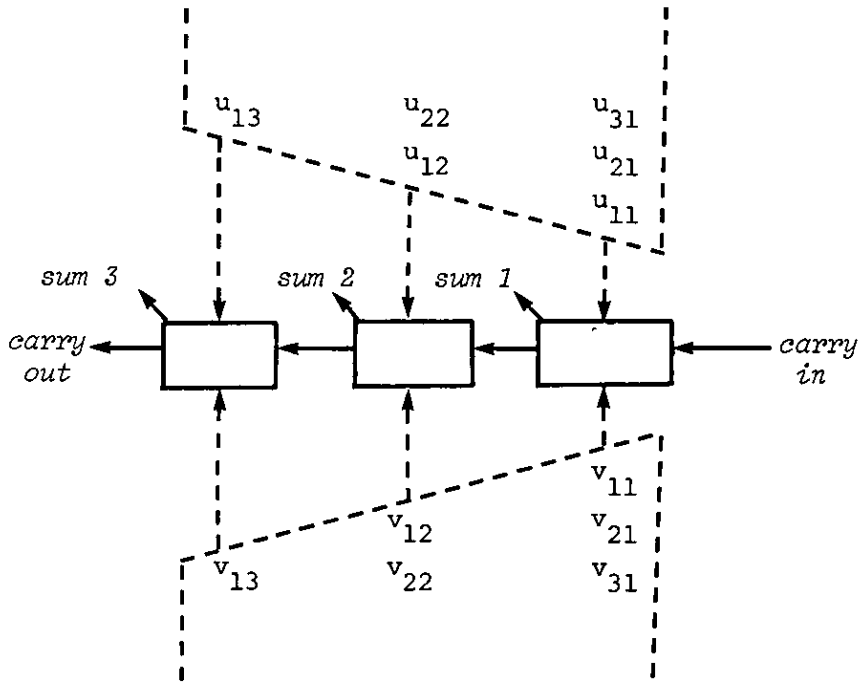


Figure II.B.2.-f1: A Pipelined Integer Adder (with $k=3$).

input lines and then outputs the *carry* through the output line. It is relatively easy to check that when the pair (u_{ij},v_{ij}) enters a module, then at the same time-step the carry, required to produce the correct i^{th} digit in the $\vec{U}+\vec{V}$ operation, enters the same module. Therefore, the pipelined adder can compute each sum of U_i+V_i in every cycle in the steady state. This algorithm can also be applied to *SIMD* computers but without considering the binary representations of the digits.

A significant requirement of parallel processor systems, that is rarely a requirement for sequential systems, is the necessity for rearranging data

in order to take advantage of the opportunities for parallelism. Two-dimensional matrix calculations are particularly susceptible to these requirements. In particular for matrix multiplication, it is necessary to align the rows of one matrix with the columns of another to obtain maximum efficiency; according to this, the computation of the *A.B* matrix product requires different types of accessing capabilities, than the computation of the *B.A* matrix product, since in the former case the *rows* are required, whilst in the latter case the *columns* of matrix *A*. However, for complete flexibility, one should be able to have simultaneous access to both rows/columns of a matrix, consequently the data had to be arranged, in some fashion, to permit such an access. A number of solutions were proposed to this problem.

One way, as already has been mentioned earlier (see *par.-I.B.3.2.1*), was to consider a special, but not simple, unconventional storage scheme for the matrix, known as a *Skewed storage*; according to this scheme, the rows of the matrix are stored so that successive elements of each row are in successive processors, which in turn implies that the elements of a column will not be stored in a single processor, but in successive processors. Thus, it became possible to fetch either a row or a column in a single memory cycle.

However, sometimes, as in the case of the matrix product algorithm, one may actually need to *transpose* a matrix, while this may never be required on a sequential computer. A solution to this problem was to build an efficient mechanism, in the form of an interconnection pattern, for obtaining the matrix transpose. This pattern is called the *Perfect Shuffle* (see Stone [*STON71*]). Since 'bidirectional' interconnections are normally not much more expensive that 'unidirectional' interconnections,

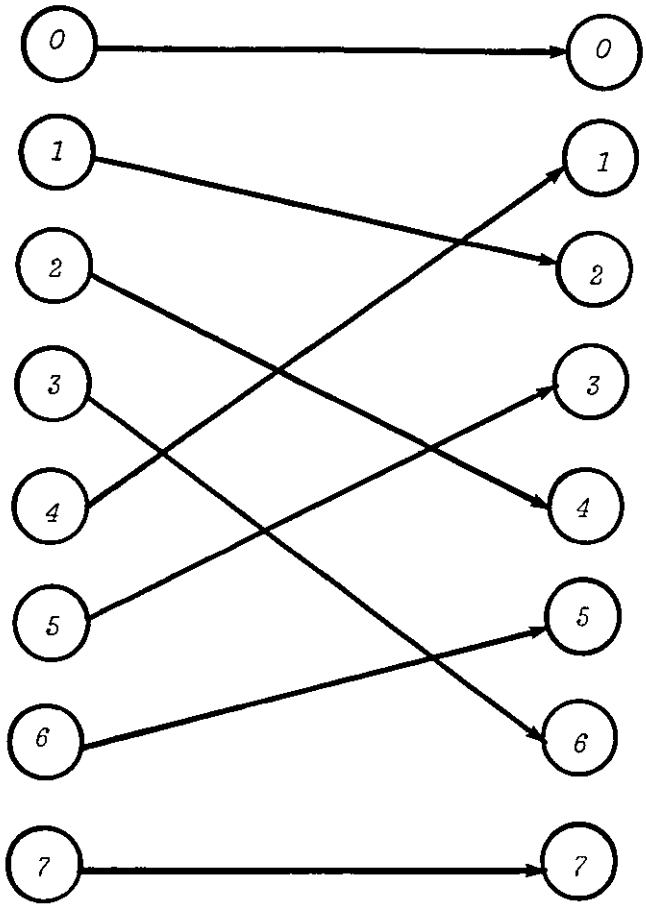
one can assume that the *Inverse Perfect Shuffle* is available, if the perfect shuffle is implemented; this is the permutation obtained by reversing the arrows in *Figure (II.B.2-f2)*.

There is a number of parallel algorithms that can make effective use of the perfect shuffle or its inverse, such as, algorithms to perform Fourier Transforms, Sorting, etc.; however, in some applications one of the two is favoured over the other and so, for greatest flexibility, it appears to be advantageous to have both available.

In concern with the number of shuffles required when utilizing such an interconnection pattern, as an example, a perfect shuffle interconnection pattern, installed in an *ILLIAC*-like computer, can transpose a matrix of size $\sqrt{n}\times\sqrt{n}$ by $log_2 n$ perfect shuffles, when $n=2^m$ (m is integer). In the case of a *Pipelined Vector* computer - e.g. *CDC STAR 100* computer, it would require $2log_2 n$ passes, since it computes the perfect shuffle of a vector in two passes; however, in such a type of computer the *skewed* storage scheme cannot be applied, for it does not have the equivalent of index registers in each processor (as for example, the *ILLIAC IV* computer) and therefore this type of computer has a bias to algorithms that process matrices only by rows or only by columns.

As we have mentioned earlier (see *par.-II.B.1*), inefficient sequential algorithms may lead to the most efficient parallel algorithms and even more emphatically, sequential algorithms that are apparently inherently sequential may have hidden a great deal of parallelism. This fact has led to the development of some powerful techniques to exploit such a hidden parallelism and produce efficient parallel algorithms.

One of these techniques is the *Recursive Doubling* technique, so-called because it divides the original computation into two independent



Note: The processors on the left are the same as those on the right; they appear twice for clarity.

Figure II.B.2-f2: The *'Perfect Shuffle'* Interconnection Pattern of Eight Processors.

smaller computations of equal complexity, each capable of being simultaneously executed on a *SIMD* computer; these in turn are subdivided into smaller tasks, recursively, but after each stage of a computation the intermediate expressions double in their complexity. To exemplify this technique let us consider the recurrence problem:

Given coefficients: a_1, b_1, for $1 \leq i \leq n$

$$x_0 = 1,\ x_1 = b_1$$

$$x_i = b_i x_{i-1} + a_i x_{i-2}\ ,\ \text{for } i \geq 2; \qquad (II.B.2:3)$$

compute x_i, for $2 \leq i \leq n$.

Although there is no associative operator apparent in this case,

the problem can be reformulated as a vector-matrix problem with an associative operator.

Let $X_i = [x_i \; x_{i-1}]^T$ and $A_i = \begin{bmatrix} b_i & a_i \\ 1 & 0 \end{bmatrix}$, then

$$X_i = A_i . X_{i-1} \qquad (II.B.2:4)$$

is the recurrence relation satisfying the problem.

In order to generate X_i from X_1, all that is required is to pre-multiply the latter by a series of (2×2) matrices as follows:

$$X_i = \left(\prod_{j=2}^{i} A_j\right) . X_1 \; , \qquad (II.B.2:5)$$

where all of the A_j, $j=2,\ldots,n$ are known explicitly. The formula *(II.B.2:5)* can then be evaluated by utilizing the recursive doubling, or otherwise *log product* technique. Because of the associativity of matrix multiplication it is possible to construct all of the products:

$$\prod_{j=2}^{i} A_j \; , \; i=2,\ldots,n, \qquad (II.B.2:6)$$

in the manner shown below.

Let us assume for convenience that $n=2^k+1$, so that, there are 2^k matrices A_j and for illustrative purposes that $k=2$; then we have:

$$\begin{bmatrix} A_2 \\ A_3 \\ A_4 \\ A_5 \end{bmatrix} \cdot \begin{bmatrix} I \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} = \begin{bmatrix} A_2 \\ A_3A_2 \\ A_4A_3 \\ A_5A_4 \end{bmatrix} , \begin{bmatrix} A_2 \\ A_3A_2 \\ A_4A_3 \\ A_5A_4 \end{bmatrix} \cdot \begin{bmatrix} I \\ I \\ A_2 \\ A_3A_2 \end{bmatrix} = \begin{bmatrix} A_2 \\ A_3A_2 \\ A_4A_3A_2 \\ A_5A_4A_3A_2 \end{bmatrix} . \qquad (II.B.2:7)$$

Then, the final vector of matrices can be used to compute all of the X_i simultaneously, by taking the product of each component of this vector with X_1.

In general, for $n=2^k$, there are $\log_2 n=k$ vector operations, the i^{th}

component of which is of length $n-2^{i-1}$. The obtained speed-up ratio is proportional to $n/log_2 n$, which makes this technique well-suited to Parallel computers with infinitely many, or at least n processors. It is worthwhile noting that as the construction of the vector products progresses, the extent of parallelism in the products decreases by an amount which is a power of two. This technique, obviously, can be extended to linear recurrences of all orders, as well as, to some non-linear recurrences, e.g. spline functions, linear ordinary differential equations with non-constant coefficients, etc.

In general terms, algorithms having adopted the above technique can be given the form of an evaluation tree (see Heller [*HELL78*]). Let us consider, as a simple example, the evaluation of the expression:

$$A_n = a_1 \circ a_2 \circ \ldots \circ a_n , \qquad (II.B.2:8)$$

where $\circ$ is any associative operator. Applying the recursive doubling technique to this expression, produces an algorithm that is illustrated by the evaluation tree in *Figure (II.B.2-f3)*, where at each level the operations are independent and identical and so, may be executed simultaneously.

The first level has the greatest number of operations being $\lceil n/2 \rceil$[†], which means that $\lceil n/2 \rceil$ processors will be sufficient to evaluate the operations at each level simultaneously. The number of levels is $\lceil log_2 n \rceil$ and so by utilizing $\lceil n/2 \rceil$ processors the result A_n may be evaluated in $\lceil log_2 n \rceil$ time-steps. Heller called this algorithm the associative *fan-in* algorithm, but it is more familiarly known as the *log sum* and *log product*, when the operators are + and ×, respectively.

[†] *As $\lceil x \rceil$ is defined the least integer greater than or equal to 'x'; as $\lfloor x \rfloor$ is defined the greatest integer less than or equal to 'x'.*

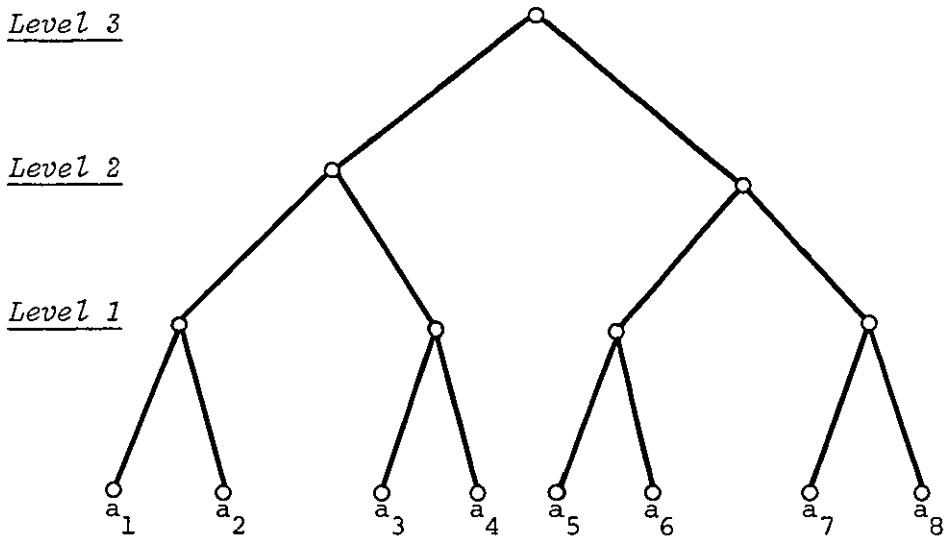


Figure II.B.2-f3: The Evaluation Tree of the Expression A_n (for $n=8$).

A special case of the fan-in algorithm is the *Inner* or *Scalar Product* of two n-vectors $\vec{x}$ and $\vec{y}$, which has the form:

$$\sum_{i=1}^{n} x_i y_i \ , \qquad (II.B.2:9)$$

where $\vec{x}=(x_1,x_2,\ldots,x_n)$ and $\vec{y}=(y_1,y_2,\ldots,y_n)$.
In this case, obviously the n products are independent and therefore can be evaluated simultaneously, utilizing n processors; then, it follows a *log sum* and the result is obtained in $\lceil log_2 n \rceil + 1$ time-steps. This is illustrated in *Figure (II.B.2-f4)*.

In accordance with this, the matrix product given by the formula *(II.B.2:2)* can be similarly evaluated, since it consists of $n \times m$ independent inner products and consequently the matrix C can be evaluated in $\lceil log_2 p \rceil + 1$ time-steps utilizing $n \times m \times p$ processors.

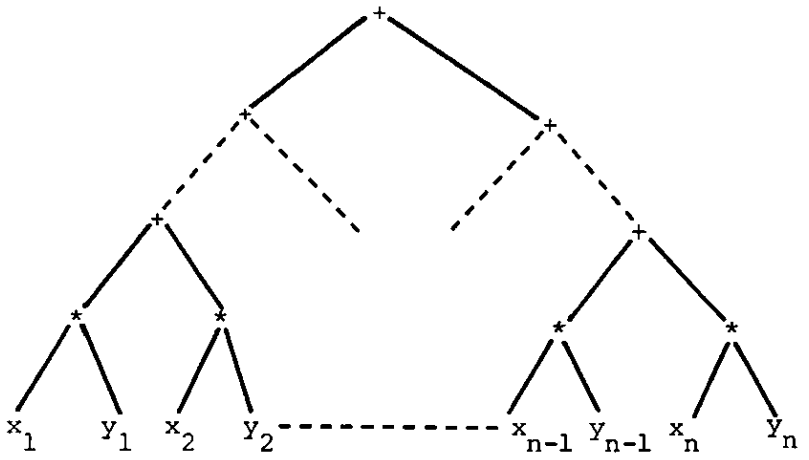


Figure II.B.2-f4: The *Inner* or *Scalar Product* of two n-vectors $\vec{x},\vec{y}$.

Another powerful technique, known for transforming a serial computation into a highly parallel one, is the *Cyclic Odd-Even Reduction* technique; although this technique is quite different from recursive doubling, it appears to be applicable to the same class of problems. This technique will be discussed in more detail in *Chapter V*, since it was extensively used to solve tridiagonal linear systems of equations on the *NEPTUNE* (*MIMD*) parallel processor system.

But the numerical area was not the only area to be searched and algorithms and techniques to be proposed for, as suitable for *SIMD* and *Pipelined* computers. Wyllie [*WYLL79*], in 1979, presented some non-numerical algorithms mainly applied to various data structures; for example, using a special technique called *doubling*, he considered an algorithm to count the number of elements in a linked list and another to delete an element from a linked list.

The *sorting* of a number of keys has been another interesting non-numerical problem, and the most widely known algorithm to tackle it, utilizing a linear processor array, is the *Odd-Even Transposition Sort*

(see Knuth [*KNUT73*]). The algorithmic procedure starts with the storing of the n keys, to be sorted, let us assume, in ascending order, in the linear processor array. The problem can be solved in n time-steps. The *odd*- and *even*-numbered processors are activated alternately. In each cycle, the comparison-exchange operations are performed as follows: The key, in every activated processor, is compared with the key stored in its right-hand neighbouring processor, and the one with the smaller value is stored in the activated processor. Hence, in n cycles the keys will have been sorted in the linear processor array, as is exemplified in *Figure (II.B.2-f5)*.

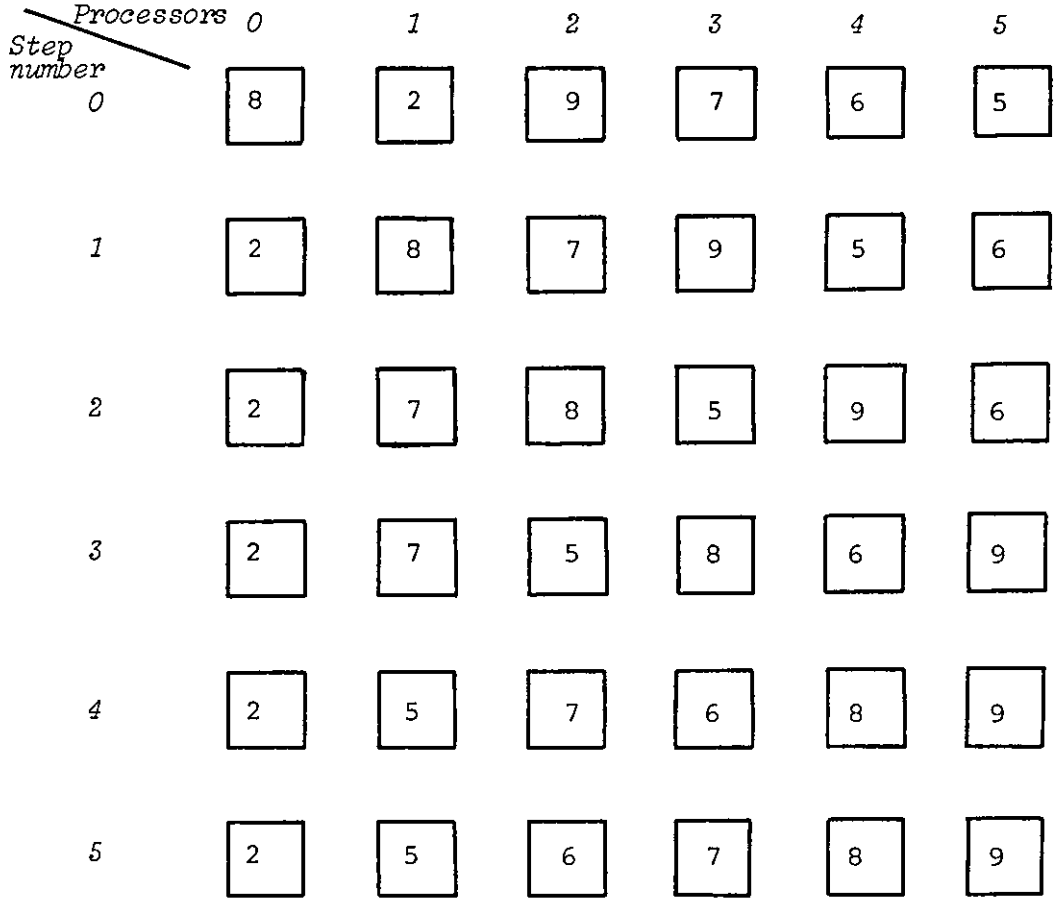


Figure II.B.2-f5: The '*Odd-Even Transposition Sort*' on a Linear Array of Processors (The *even*-numbered processors have been activated first).

A partial conclusion, which can be drawn from all the above mentioned algorithms, is that, when one designs and develops a parallel algorithm, unrealistically, considers that it is going to be implemented on a parallel architecture, that can always provide as many processors as required by the computation, to achieve its minimum execution time; a so-called *unlimited architecture*. The computations constructed for such unlimited architectures are then called *unlimited computations*.

Then, a second, more realistic and practically of equal efficiency, algorithm is obtained, which will be implemented on the actually existing parallel processor architecture, with a maximum of p processors, which is called a *p-limited architecture*.

There are two fundamental principles to construct the processors limited algorithms, the *algorithm-decomposition* and the *problem-decomposition* principles (see Hyafil and Kung [*HYAF74*]).

According to the former principle, if the originally unlimited computational algorithm performed q_i operations during a step i, then provided it has a maximum of p processors, $\lceil q_i/p \rceil$ time-steps will be required to perform the same step i.

On the other hand, according to the latter principle, the original task is partitioned into smaller tasks of order equal to the number of the available processors and the parallel algorithm is applied to each of them.

A simple example of these principles is the generalization of the *Odd-Even Transposition Sort* discussed earlier, so that, each processor, instead of a single key, will hold a sorted subsequence of keys. In this case, the algorithm will be utilizing a *merge-splitting* operation, instead of the comparison-exchange one; the optimal speed-up will be

achieved in cases where the number of keys to be sorted is large, relative to the number of linearly connected processors, since, this way, each processor will have a large amount of computing to perform, compared to the interprocessor communication, which relatively minimizes the latter, otherwise significant, overheads.

In a similar way, one may consider the evaluation of the expression A_n given in *(II.B.2:8)*, when $p \leq n/2$ processors are available; then, to evaluate the computations (e.g. *sums*) of the first level, it would require at most $\lceil n/2p \rceil$ time-steps, of the next higher level at most $\lceil n/4p \rceil$ time-steps, and so on, until the last level, which would require at most $\lceil n/2^{\lceil \log_2 n \rceil} p \rceil$ time-steps. Consequently, the total required run-time will be:

$$\begin{aligned} T_{total} &\leq \lceil n/2p \rceil + \lceil n/4p \rceil + \ldots + \lceil n/2^{\lceil \log_2 n \rceil} p \rceil \\ &< (1+n/2p) + (1+n/4p) + \ldots + (1+n/2^{\lceil \log_2 n \rceil} p) \quad \text{(since } \lceil x \rceil < 1+x\text{)} \\ &= \lceil \log_2 n \rceil + n/p(1/2 + 1/4 + \ldots + 1/2^{\lceil \log_2 n \rceil}) \qquad \textit{(II.B.2:10)} \\ &\leq c_1 . \log_2 n + c_2 . n/p \\ &= O(\log_2 n + n/p) \ . \end{aligned}$$

In the case that $p = n/\log_2 n$, the T_{total} tends to become of $O(\log_2 n)$ and so, an execution time of similar order, as the previous algorithm, can be achieved by utilizing fewer processors (reduced by a logarithmic factor).

In conclusion, numerous other algorithms have been developed for *SIMD* and *Pipelined* computers, utilizing the fundamental schemes, techniques and principles mentioned earlier, covering various fields of science. To selectively refer to some of them, Gilmore [*GILM71*], Liu [*LIUJ74*], Hayes [*HAYE74*] and Sameh, Chen and Kuck [*SAME74*], proposed algorithms to

solve systems of equations arising from differential equations; Smith [*SMIT71*], investigated *cellular* algorithms to perform pattern recognition, Levitt and Kautz [*LEVI72*], algorithms to solve graph problems, Thompson and Kung [*THOM77*], studied sorting algorithms and Barlow, et al [*BARL82a*], algorithms for eigenvalue problems.

The majority of these algorithms have explored perfectly the parallel hardware potential of the available *SIMD* and *Pipelined* computers, despite the unavoidable overhead constraints imposed by the large amounts of synchronization and interprocessor communication.

II.B.2.1: PARTICULAR CONCEPTS AND PERFORMANCE FEATURES OF THE *'DAP'* SYSTEM

To be consistent with the promises made in *Chapter I*, we shall, briefly, refer to some of the particular characteristics of this specific *SIMD* parallel processing system, accessible from Loughborough University via a *modem*.

The language for programming the *DAP* system came as a result of the *'evolutionary school of thought'* (see Parkinson [*PARK82*]), who transformed the abysmal normal *FORTRAN* into an efficient and more user friendly parallel language, the so-called *DAP FORTRAN*[†].

Although, it is out of the scope of this Thesis to present the language features here (see *ICL DAP* Manual [*DAPM78*]), we shall emphasize on a couple of fundamental powerful facilities that distinguish this issue of *FORTRAN* from the normal one.

The first powerful facility was introduced under the form of an extended list of *data modes*; in other terms, whilst normal *FORTRAN* deals

[†] *It has many features which make it specific for the 'DAP' system.*

only with *scalar* variables (or sets of them), the *DAP FORTRAN* issue utilizes two new modes - *vector* and *matrix*. A *DAP FORTRAN* vector is a data structure with *64* values; a *DAP FORTRAN* matrix is a data structure of *4096* values, considered either as a *(64×64)* array or as a long vector.

Variables can still be of *integer*, *real*, *logical* or *character* type [†] and one can define sets of vectors or matrices; however, there is a distinct logical difference between *64* vectors and a matrix, albeit the equivalent (*4096*) number of values.

Another powerful facility, which occurs when one wishes to provide *conditional* computations, is the utilization of the *logical expression* in a fashion similar to an *index*; for example, the operation, which requires an *IF* statement in normal *FORTRAN*, here appears

$$X(Y.GE.\emptyset) = SQRT(Y)^{\ddagger}, \qquad (II.B.2.1{:}1)$$

where a value *FALSE* or *TRUE* is obtained, each time, from the logical expression-*index* '*Y.GE.Ø*'. This construct is called *LOGICAL* or *MASK INDEXING* and is a powerful construct, since it allows the programmer close access to the computations, which is an essential programming necessity for optimal utilization of *SIMD* systems.

In general, more important than understanding how an algorithm works, is the understanding of the time dependence of the algorithm. For example, if a sequential *FORTRAN* code has three nested *DO*-loops, each performed n times, then, that algorithm requires a time proportional to n^3 on a sequential computer; the corresponding parallel algorithm can only have a single loop and so requires a time proportional to n, provided the system comprises at least n^2 processing elements.

[†] *As in normal FORTRAN.*

[‡] *It computes the square root of every member of Y set, provided it is positive.*

The *Speed-up* ratio of the parallel computer to any given serial computer is therefore a function of the size and complexity† of the problem, with the parallel computer giving its best performance when the number of processing elements generally matches the problem size.

On the other hand, to measure the *Efficiency* of a parallel algorithm, one needs to consider both, its complexity, as well as, its cost in terms of the number of *PEs* utilized. The 'Efficiency of Processor Utilization' - *EPU*, is defined with respect to the parallel algorithm and the fastest known sequential algorithm‡ for the same problem. In particular, for a problem *Pr* and its parallel algorithm *Pa*, we define:

$$\mathrm{EPU(Pr,Pa)} = \frac{\textit{Complexity of the fastest sequential algorithm for Pr}}{\textit{Number of PEs utilized by Pa} * \textit{Complexity of Pa}} .$$

(II.B.2.1:2)

In a general terms review, the overall performance of a *DAP FORTRAN* program is highly application dependent; in other words, it depends upon several factors, such as the mode of the data processed by the program, the type of operations performed upon the data, and the degree of parallelism in the algorithm chosen. In respect to the definition of the efficiency in terms of processor utilization and the fact that at each substage of the algorithm half of the processors are eliminated, experience has shown that it is not catastrophic to continually switch processing elements off; in fact, the highest performance of the *DAP* system, relative to serial systems, occurs in problems with many logical operations, rather than problems in which all the *PEs* are apparently fully occupied - e.g. a dense matrix multiplication does not exhibit a peak performance.

---

†*More details about this term in paragraph (II.B.3).*

‡*Since the lack of a 'true' meaning of the 'T_{serial}' time on a bit-organized computer system.*

Finally, in respect of the various data modes, they are mapped onto the *DAP* store in a different way, so that individual processing elements have different access to their components. In particular, a *vector* component is processed by *64* cooperating processing elements, since each bit of the component is kept in the local store of a different processing element; whilst a *matrix* component, since it is entirely within the local store of a processing element, is processed by this individual element. A *scalar* may be processed either within the array or in the array 'Master Control Unit' - *MCU*.

Arithmetic operations are performed most efficiently on matrices. To compare the time required for *vector* and *matrix* processing, although the former operation is faster per component, its overall efficiency is less, since a matrix operation processes *64* times as many components. Although *scalar* processing is still much faster than an operation on a matrix component, the matrix operation will process *4096* components simultaneously. In the case that a large amount[†] of unavoidable *scalar* processing occurs, it should be performed within the *host* section of a program, that is on the *host-2900 ICL* computer.

In conclusion, to further clarify the above statements, in brief, a *DAP* program has two sections, the *host* and the *DAP* section. Since *DAP FORTRAN* has no input/output facilities and all calls, from the *host* section to the *DAP* section, are parameterless subroutine calls, data is passed between them via named *COMMON* blocks; these *DAP FORTRAN COMMON* blocks are kept in the *DAP* store, but are accessible to both, the *DAP* and the *host*.

[†]*A certain amount of 'scalar' processing can be performed by the 'DAP' facilities, avoiding the overheads of a return to the 'host' computer.*

II.B.3: Fundamental Algorithm Structural Concepts To Exploit The Potential Of MIMD Computer Architectures

Although *Multiprocessor* (*MIMD*) architectures have now been in existence for several years (see the *D825*, Anderson, et al [*ANDE62*], in fact dates back to the early 60's), relatively very little has been published so far about designing efficient parallel algorithms for this type of system.

A principal clarification which must be made is that the categorization of algorithms to *Synchronized* and *Asynchronous* (see *par.-I.B.1*) aimed, mainly, to distinguish the algorithms in respect to the parallel computer systems particular features; however, the greater flexibility of *Multiprocessor* architectures, as being composed of sets of independent processors, makes both categories of algorithms efficiently applicable on them, with the latter posing more difficulties in the respect that they can allow asynchronous processes.†

The major issue for the *Multiprocessor* architectures is that of partitioning a problem into many processes that can be executed in parallel onto them. For a small number of processors, say two to four, this problem is not a significant one; but, for several processors, say sixteen (e.g. the *CYBA-M* at UMIST), or more, the problem becomes extremely difficult. Programs without a specific iterative structure are seldom so complex that they can have sixteen to thirty-two distinct sub-processes. However, programs with an iterative structure are likely to be better suited to *SIMD* systems and will execute with somewhat lower efficiency on *MIMD* systems due to resource allocation and synchronization overheads.

---

† *This fact justifies the use of the term 'Asynchronous Multiprocessor' for this type of system.*

In general terms, a parallel algorithm, for a *Multiprocessor* architecture, can be defined as a collection of concurrent processes that may operate simultaneously for solving a given problem. To ensure that an algorithm performs correctly and utilizes parallelism effectively for solving a given problem, it is usually necessary to have *interactions* amongst the processes; in other terms, in the program which controls a process there may be some points where the process can communicate with other processes, the so called *interaction points*, which divide a process into *stages*. Thus, at the end of each stage, a process may communicate with other processes before starting the next stage; but, this communication should be minimal (i.e. tasks should be as independent from each other as possible), while the locality of the tasks should be maximal (i.e. all the data accessed by a process/processor should be kept in its local memory).

The time required by a fixed stage of a process is usually not a constant, since there are some major sources for causing *fluctuations* in process speed. With respect to the *Multiprocessor* architecture, these *fluctuations* in process speed are due to the various processors speeds, in addition to the probable asynchronous behaviour of them and the arising memory conflicts. Such *fluctuations* are also due to the Operating System scheduling policies, that assign certain processors to perform I/O, allocate processors to processes, switch a processor from one process to another, and so on. Finally, the process speed may be influenced by the user environment itself, since the amount of resources allocated to a particular process, at a given time, is a variable depending upon the number of created processes by the user and their priorities.

On the other hand, the work to be performed by an algorithm may depend on its input instances; consequently, the work performed by a stage is unpredictable, since, in general, the property of the input to a stage is unpredictable or regarded as such. Therefore, the time required by a stage can vary in an unpredictable way, a fact which can easily make us assume that it is a random variable satisfying some distribution function, which hopefully under certain circumstances can be estimated.

But, before we continue further on, we should tidy up in a consistent way the two, basically, distinguished approaches for tackling an occurring problem, the *synchronous* and the *asynchronous* approach.

In the former approach, one *synchronizes* the processes forcing them to wait[†] for the required inputs, whilst in the latter approach lets them continue *asynchronously*. The motivation for the *asynchronous* approach resulted from the fact that, in some cases, no *synchronized* parallel algorithm would load all the processors equally, in addition to the performance degradation due to the required synchronization. However, the *asynchronous* behaviour of an algorithm may lead to serious issues regarding its correctness and efficiency. The former issue arises because during the execution of an algorithm, operations from different processes may interleave in an unpredictable manner; whilst, the latter issue arises because any synchronization introduced, for correctness purposes, requires an extra time and also reduces concurrency.

An attempt to strictly define a *synchronized* parallel algorithm would result in that, 'It is an algorithm consisting of processes explicitly

---

[†] *Thus, theoretically, there is no way to avoid the possibility of one process sitting 'idle' for a potentially large (could be infinite?) amount of time, before other processes finish generating the operands it requires.*

utilizing synchronization primitives[†], in which, upon completion of a stage, a process may have to wait for the results of other processes before resuming its execution'; in such an algorithm a task is decomposed into subtasks, of the same, hopefully, size, so that each subtask is solved by one process of the algorithm.

However, one must bear in mind that it is not always advantageous to create as many processes as possible, according to the maximal decomposition of a task, since the execution time of the synchronization primitives is usually non-negligible (e.g. a typical execution time for these primitives is usually in the order of a couple of hundreds of additions). Also, since the time required by a stage of a process is a random variable, *synchronized* algorithms have the drawback that some processes may be 'blocked' at a given time (e.g. while waiting for a signal which is supposed to be issued by some *'dead'* process). A characteristic example of a *synchronized* algorithm is the *producer-consumer* type of program.

On the other hand, an *asynchronous* parallel algorithm can be defined as, 'An algorithm consisting of processes which communicate amongst themselves only through the use of some *global* variables (possibly updated within a critical section to ensure logic correctness) or *shared* data; and, at the completion of a stage, a process either terminates or proceeds further, without any delay, according to the current contents of the global variables and the results just obtained from the last stage'. The *asynchronous* characterization is due to the fact that synchronizations are not required to ensure that specific inputs are available for processes at various times; however, there still exists the possibility

[†]*Synchronization primitives are required for synchronizing processes and implementing critical sections.*

of processes being 'blocked' from entering critical sections, since the access to them follows the 'First-In, First-Out' (*FIFO*) priority rule.

In the following, we shall, briefly, exemplify the above fundamental algorithm structural concepts, by recalling some of the previous examples, considered in (*par.-II.B.2*), and adapting them to map onto *Multiprocessor* architectures.

To start with, let us consider the expression of (*II.B.2:8*):

$$A_n = a_1 \circ a_2 \circ \ldots \circ a_n \ , \qquad (II.B.3:1)$$

which can be evaluated on *SIMD* systems utilizing the associative *fan-in* algorithm. It was obvious from the given *Figure (II.B.2-f3)* that the operations at each level are independent, which is not true for the operations amongst the levels. Therefore, the above expression can be easily decomposed into a number of independent processes, to be carried out by several processors simultaneously, with a minimum amount of interference amongst them.

Consequently, on a hypothetical *Multiprocessor* system of p processors, the above expression can be partitioned into the following p subsets:

$$A_n = (a_1 \circ a_2 \circ \ldots \circ a_\ell) \circ (a_{\ell+1} \circ a_{\ell+2} \circ \ldots \circ a_{2\ell}) \circ \ldots \circ (a_{(p-1)\ell+1} \circ a_{(p-1)\ell+2} \circ \ldots \circ a_{p\ell}) \ , \qquad (II.B.3:2)$$

where $\ell = \lceil n/p \rceil$. The evaluation tree for this algorithm is presented in *Figure (II.B.3-f1)*.

Before we proceed to the analysis of this algorithm, we should clarify something already utilized in (*par.-II.B.2.1*), the *complexity*

concept. In general terms, when measuring the cost of executing a program, one customarily defines a *complexity* (or *cost*) function ϕ, where $\phi(n)$ is either a measure of the time required to execute the algorithm of a problem of size n, or is a measure of the memory space required for such an execution. Accordingly, one may speak of either the *time-complexity*† or the *space-complexity* function of the algorithm, or refer to either of them as simply a *complexity* (or *cost*) function of the algorithm.

In this Thesis our principal concern will be with *time-complexity* functions, but we shall distinguish, where appropriate, between the *time-complexity* and the *computational-complexity* of an algorithm. The latter term will be utilized to refer to estimates of the computational power required to solve a given problem, being measured by the number of *arithmetic* or *logical* operations required. Within our context, the *computational-complexity* measure will be considered as a branch of the *time-complexity* measure.

A further clarification in general is that, in particular for the analysis of the *computational-complexity* of numerical algorithms, it would be convenient to distinguish two further branches of it, (i) *algebraic* and (ii) *analytic-complexity* measures.

The objectives of *algebraic-complexity* studies are manifold: e.g. (1) to find out how many arithmetic operations are used by a given algorithm, (2) how many arithmetic operations are required to solve a given problem, and (3) what is the best way to solve a given problem in terms of arithmetic operations. The solution of linear sets of equations

---

†*There are, apparently, two 'time-complexity' functions for problems, defining the 'lower' and 'upper' (time-complexity) bounds, and the ultimate goal is the coincidence of these functions to produce the 'optimal' algorithm; however, for most of the problems this goal is not yet realized (see [WEID77]).*

by direct methods is an example of a problem studied in terms of *algebraic-complexity*.

Analytic-complexity, on the other hand, addresses the question of how much computation has to be performed to obtain a result with a given degree of accuracy, and focuses on computational processes which in a certain sense never end. Iterative processes provide an obvious example.

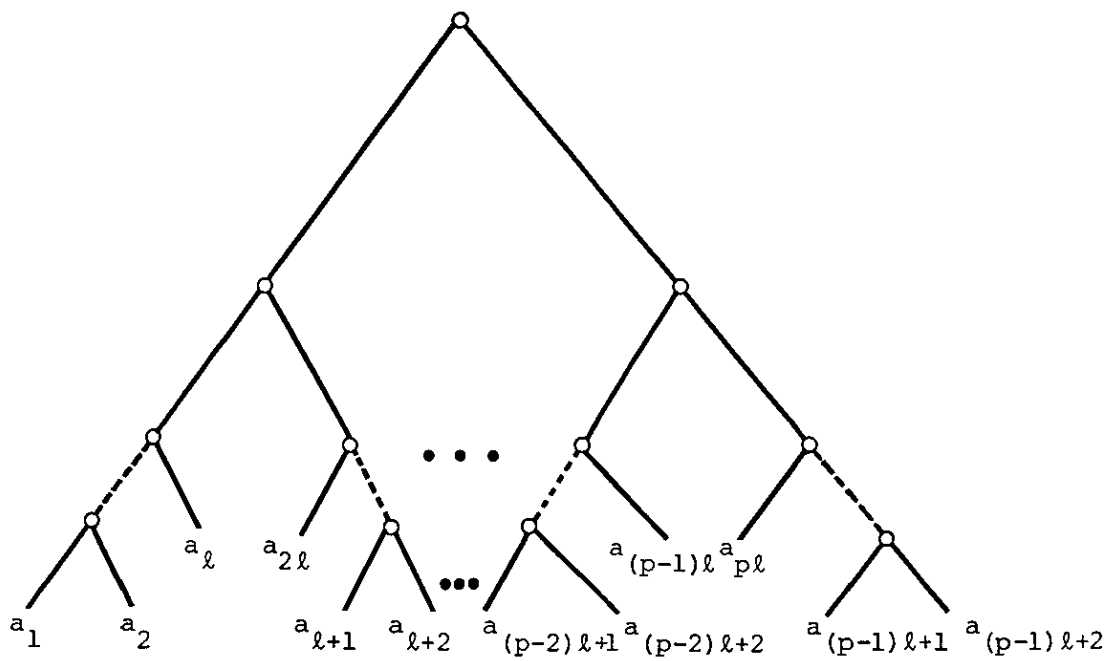


Figure II.B.3-f1: The Evaluation Tree of the Expression A_n of *(II.B.3:2)* (in the case that p is even).

Back to the previously presented algorithm by *(II.B.3:2)* and in order to estimate the speed-up and efficiency ratios, it is required to estimate the sequential T_s and the parallel T_{nop} [†] running times. Thus, the *time-complexity* for sequential execution is:

$$T_s = n\text{-}1 \text{ time-steps} , \qquad (II.B.3:3)$$

whilst in parallel execution (with p processors) is:

[†] $T_{\underline{no}.\ of\ processors}$.

$$T_p = \lceil n/p \rceil - 1 + \lceil \log_2 p \rceil \text{ time-steps,} \qquad (II.B.3:4)$$

where $\lceil log_2 p \rceil$ are the final time-steps to estimate the values of the p-paths.

The *Speed-up* and *Efficiency* ratios then are:

$$S_p = \frac{T_s}{T_p} = \frac{n-1}{\lceil n/p \rceil - 1 + \lceil \log_2 p \rceil}$$

$$\leq \frac{n-1}{n/p-1+\log_2 p} \quad \text{(since } \lceil x \rceil \geq x\text{)}$$

$$= p - \frac{p\log_2 p - p + 1}{n/p-1+\log_2 p}$$

$$= p\left[1 - \frac{p\log_2 p - p + 1}{n-p+p\log_2 p}\right] , \qquad (II.B.3:5)$$

and $$E_p = \frac{S_p}{p} \leq 1 - \frac{p\log_2 p - p + 1}{n-p+p\log_2 p} ; \qquad (II.B.3:6)$$

in the case that $n>>p$ the fraction in *(II.B.3:6)* tends to zero and consequently the *Speed-up* and *Efficiency* ratios approximate their optimal theoretical values of p and *one*, respectively. To the contrary, for a fixed value of n, an increase of the number of processors implies a decrease of the speed-up and efficiency, due to various occurring overheads.

The same strategy of decomposition can now be applied to the *Inner* or *Scalar product* of two n-vectors of *(II.B.2:9)*. Assuming there are p processors, then the p subsets, of size $\ell = \lceil n/p \rceil$, have independent computations and can be evaluated simultaneously. The corresponding evaluation tree will be of the same form, as that of the expression *(II.B.3:2)*, depicted in *Figure (II.B.3-f1)*. This algorithm proceeds by evaluating the multiplications and additions of all ℓ subsets concurrently, to conclude, with the addition of the values obtained from

the ℓ subsets, in $\lceil log_2 p \rceil$ time-steps. The *time-complexity*[†] of this algorithm, for sequential and parallel execution respectively, is given by:

$$T_s = 2n-1 \text{ time-steps (i.e. n-multiplications+(n-1)-additions)}, \qquad (II.B.3:7)$$

and $$T_p = 2\lceil n/p \rceil - 1 + \lceil \log_2 p \rceil \text{ time-steps}. \qquad (II.B.3:8)$$

Consequently, the *Speed-up* ratio is:

$$S_p = \frac{T_s}{T_p} = \frac{2n-1}{2\lceil n/p \rceil - 1 + \lceil \log_2 p \rceil}$$

$$\leq \frac{2n-1}{2n/p-1+\log_2 p} \quad (\text{since } \lceil x \rceil \geq x)$$

$$= p - \frac{p\log_2 p - p + 1}{2n/p-1+\log_2 p}$$

$$= p\left[1 - \frac{p\log_2 p - p + 1}{2n-p+p\log_2 p}\right] \quad ; \qquad (II.B.3:9)$$

and the *Efficiency* ratio is:

$$E_p = \frac{S_p}{p} \leq 1 - \frac{p\log_2 p - p + 1}{2n-p+p\log_2 p} \quad , \qquad (II.B.3:10)$$

where again for $n>>p$ the fraction in *(II.B.3:10)* tends to zero and consequently the *Speed-up* and *Efficiency* ratios approximate their optimal theoretical values of p and *one*, respectively.

With respect to matrix operations (i.e., addition, multiplication, etc.), the computations of the elements c_{ij} of the resultant matrix C are independent and therefore can be evaluated concurrently. For example, let us consider the addition of two $(n \times m)$ matrices, such that,

$$C = A + B \ , \qquad (II.B.3:11)$$

[†]*Since the total number of multiplications and additions, respectively remains the same, in both, 'serial' and 'parallel', executions, it is therefore convenient, for simplicity reasons, to assume the same time-step length for both.*

which is defined as: $c_{ij} = a_{ij} + b_{ij}$, for $i=1,2,\ldots,n/j=1,2,\ldots,m$.

(II.B.3:12)

Assuming again the availability of p processors, then $\lceil n/p \rceil$ rows (or columns) are assigned to each processor to carry out the operation. The *time-complexity*, for sequential and parallel execution respectively, is given by,

$$T_s = n.m \text{ time-steps}, \qquad \textit{(II.B.3:13)}$$

and

$$T_p = \lceil n/p \rceil .m \text{ time-steps}. \qquad \textit{(II.B.3:14)}$$

Consequently, the *Speed-up* ratio is:

$$S_p = \frac{T_s}{T_p} = \frac{n.m}{\lceil n/p \rceil .m} \leq \frac{n.m}{(n/p).m} = p \; ; \quad (\text{since } \lceil x \rceil \geq x) \qquad \textit{(II.B.3:15)}$$

and the *Efficiency* ratio is:

$$E_p = \frac{S_p}{p} \leq \frac{p}{p} = 1 \; . \qquad \textit{(II.B.3:16)}$$

In this case, it is apparent that both ratios are approximating their optimal theoretical values of p and *one*, respectively.

Since matrix *subtraction* and *multiplication* also imply the evaluation of $n.m$ elements of the result matrix, each of which is independent, exactly the same strategies can be applied, achieving identical speed-ups and efficiencies. One may note however that, in the case of the matrix multiplication of *(II.B.2:2)*, each component of the result matrix is a *scalar product*, and so they can be evaluated one at a time by utilizing the *scalar product* strategy discussed earlier in this paragraph.

After these simple, but characteristic, algorithm structuring examples, in the remainder of this paragraph we shall refer to some specifically significant and unique issues concerning *synchronized* and *asynchronous Multiprocessor* algorithm implementations.

One of the classical problems, which involved quite an amount of research, was the *Search for Zeros* problem. The definition of this problem is that, 'Given a continuous (or discrete) function f, having opposite signs at the endpoints of an interval of length ℓ, locate a *zero* of f within a unit interval'. Two issues of parallel algorithms were developed by Kung [*KUNG76*], one *synchronized* and one *asynchronous*, to tackle this problem, being compared with the best considered to be a known, sequential search method, i.e. the *Binary search*.

With respect to the *synchronized Zero-Searching* algorithm, it consists of k processes and at each 'iteration', each process evaluates f at one of the k points, dividing the current interval of uncertainty into $k+1$ sub-intervals of equal length. The evaluation is considered as a stage of the process. It is obvious that every iteration reduces the length of the uncertainty interval by a factor of $k+1$. This algorithm is *synchronized* in the sense that the k identical stages are synchronized, since when all of them are completed, a new uncertainty interval is computed by one of the processes.

Consequently, the algorithm will perform a total of $\lceil log_{k+1}\ell \rceil$ iterations, with an expected *time-complexity* of $\lceil log_{k+1}\ell \rceil . \lambda_k . \bar{t}$, where $\lambda_k . \bar{t}$ is the expected *time-complexity* for each iteration (rather than $\bar{t}$[†]), since one must also consider the 'penalty factor' λ_k[‡] of synchronizing k function evaluations each time. In comparison with the expected *time-complexity* of the *Binary search* algorithm, which is of $\lceil log_2 \ell \rceil . \bar{t}$ (since it takes at most $\lceil log_2 \ell \rceil$ function evaluations), the conclusion is that the *synchronized* parallel algorithm will be proved inefficient

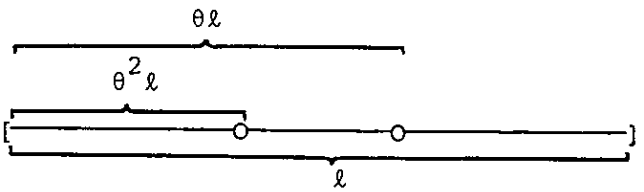
[†] *This is the 'mean' of the random time variable 't' required to evaluate 'f' at a point in the interval.*

[‡] *See [KUNG76], p.161, for the definition of this 'penalty factor'.*

for large λ_k, which usually occurs when k is large.

With respect to the *asynchronous* issue for the same problem, Kung, at first, introduced an algorithm (called AZ_2) with two processes, based on a Fibonacci law; later, he generalized this algorithm for three or more processes. These algorithms can be defined by their transitions amongst various states.

In particular for the former and simpler case, he considered two types of states, $A_1(\ell)$ and $A_2(\ell)$, which are defined by the following basic pattern:



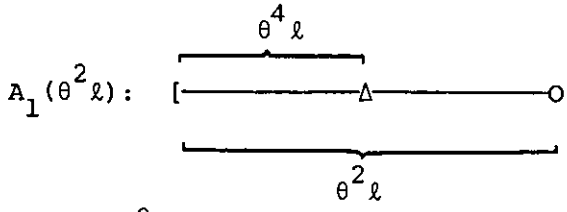
(II.B.3:17)

where $\theta^2+\theta^{\dagger}=1$; the pattern *(II.B.3:17)* is state $A_2(\ell)$ as it stands and becomes state $A_1(\ell)$ if the right inner-point is deleted. In both states the interval of uncertainty is of length ℓ, but in the former one, f is evaluated simultaneously at two points denoted by 'O', both inside the interval, whilst in the latter one, f is evaluated simultaneously at a point 'O' inside the interval and another point outside the interval.

Let us assume that we are at state $A_2(\ell)$ and that, without loss of generality, the evaluation at the left point finishes first. Then, the new interval of uncertainty is either [——————O or O——————O——————], depending upon the sign of the outcome. Suppose that the first case occurs, then the process, which just finished the evaluation at the left point, activates a new evaluation at the point 'Δ' defined as follows:

---

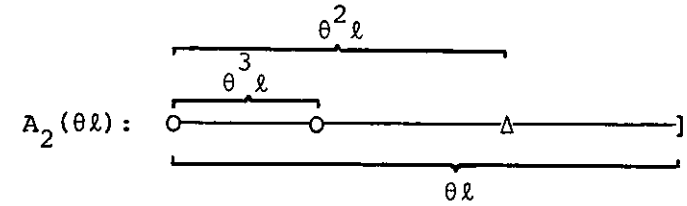
†*Namely,* $\theta=.618...$ *is the reciprocal of the golden ratio* ϕ.



(II.B.3:18)

hence, state $A_1(\theta^2\ell)$ is obtained.

In a similar way, state $A_2(\theta\ell)$ can be obtained from the other case, as illustrated by the following graph:



(II.B.3:19)

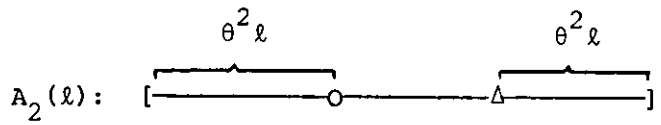
The transition to either of the above states, of state $A_2(\ell)$, is denoted by:

$$A_2(\ell) \rightarrow A_1(\theta^2\ell) \vee A_2(\theta\ell) \ . \qquad (II.B.3:20)$$

Correspondingly for $A_1(\ell)$ we have:

$$A_1(\ell) \rightarrow A_1(\theta^2\ell) \vee A_1(\theta\ell) \vee A_2(\ell) \ . \qquad (II.B.3:21)$$

The state $A_2(\ell)$, in the transition rule *(II.B.3:21)*, corresponds to the assignment of the second process at the point denoted by 'Δ' on the graph below:



(II.B.3:22)

An important property of algorithm AZ_2 is that it associates with the very simple transition tree of *Figure (II.B.3-f2)*. Assuming that the algorithm starts from state $A_1(\ell)$, it passes through all the states on one of the paths in the tree. The particular path taken by the algorithm

depends upon the input function f and the relative speeds of the two processes.

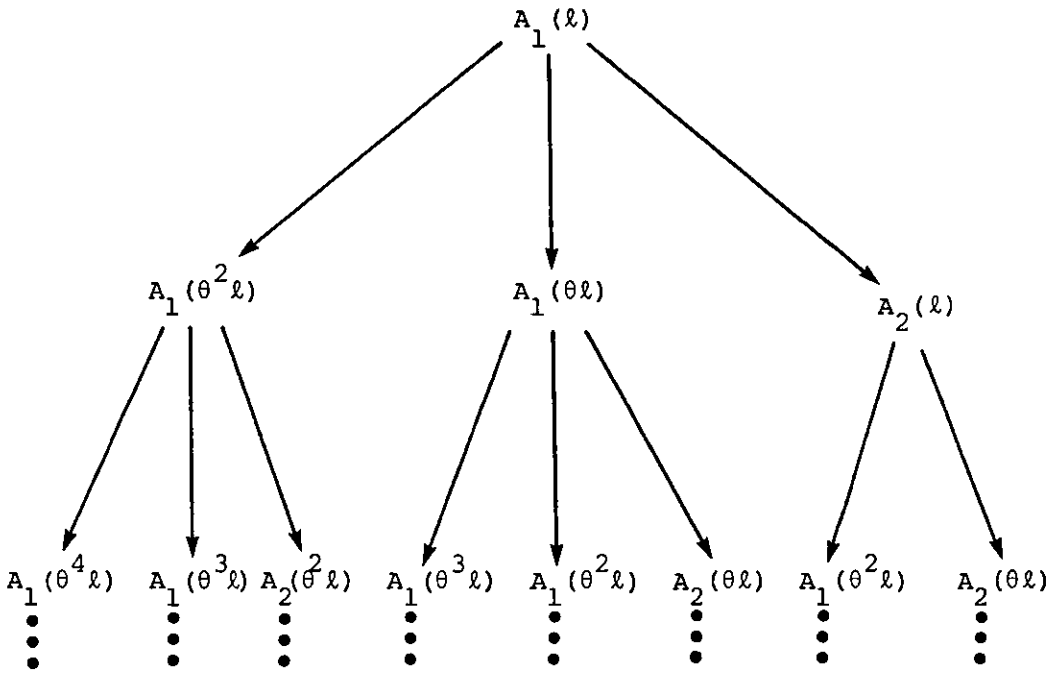


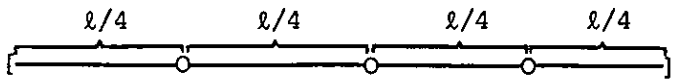
Figure II.B.3-f2: The Transition Tree of the AZ_2 Algorithm.

For the analysis of this algorithm, let n be the number of function evaluations performed by the algorithm. Since the evaluations are performed by two concurrent processes, the expected *time-complexity* of the algorithm is $\sim\frac{n.\bar{t}}{2}$, as $n\to\infty$. Consequently, the *Speed-up* ratio between the *Binary search* and this algorithm is:

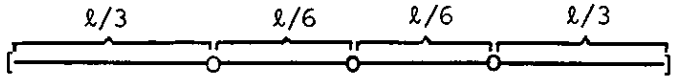
$$S_2 \sim \frac{(\log_2\ell).\bar{t}}{\frac{n.\bar{t}}{2}} = \frac{2\log_2\ell}{n}, \text{ as } n\to\infty . \qquad (II.B.3:23)$$

It is obvious from *(II.B.3:23)* that the determination of the value for n is of great interest. This value, in the 'worst' case, is given by the length of the longest path in the transition tree, in the 'best' case, by the length of the shortest path and, in the 'average' case, by the average path length.

An asynchronous algorithm with *three* processes can be similarly defined by using the following two patterns:



and *(II.B.3:24)*



.

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

An *asynchronous Zero-Searching* algorithm with k processes corresponds, in a natural way, to an *asynchronous* algorithm with $k-1$ processes for locating the maximum of an unimodal function. Thus, the patterns in *(II.B.3:24)* provide an *asynchronous* algorithm with two processes for locating the maximum of an unimodal function, which is faster than the optimal *synchronized* algorithm with two processes - (see Avriel and Wilde [*AVRI66*] and Karp and Miranker [*KARP68*]) - as long as the 'penalty factor' of synchronization is greater than one.

In the following, we shall introduce a highly important group of methods utilized to solve many numerical problems, the *Iterative Methods*. For example, zeros of a function f can be computed by the Newton iteration:

$$x_{i+1} = x_i - f'(x_i)^{-1}.f(x_i) \; ; \qquad \textit{(II.B.3:25)}$$

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

$$\vec{x}_{i+1} = A\vec{x}_i + \vec{b} \; , \qquad \textit{(II.B.3:26)}$$

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

In general terms, an *Iterative Process* can be represented by the formula:

$$x_{i+1} = f(x_i, x_{i-1}, \ldots, x_{i-d+1}) \ , \qquad (II.B.3:27)$$

and the aim is to design algorithms for *Multiprocessor* systems to speed-up the computation of the iterative process.

Basically, either of two strategies (or a combination of both) can be followed when designing *synchronized* or *asynchronous iterative* algorithms. The first one aims to exploit parallelism within the iteration function f, and the other to exploit the *fluctuations* (*a priori* considered harmful?) in process speed, mentioned earlier in this paragraph, by utilizing more than one process to compute the same function in parallel.

In a *synchronized iterative* algorithm 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 synchronized 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 *synchronized iterative* algorithms are not suitable for those iteration functions which cannot be decomposed into mutually independent tasks of the same complexity.

On the other hand, *asynchronous iterative* algorithms are parallel iterative algorithms 'free' from any synchronization restrictions. To design an *asynchronous iterative* algorithm for a general iterative process (as *II.B.3:27*), one should first identify certain variables, such that, each iterative step can be regarded as computing the new values of these variables from their old values. In general terms, it is desirable to choose these variables, such that, the updating of each of them constitutes a significant portion of the work involved in each iteration.

After these variables have been chosen, concurrent processes, which would update these variables *asynchronously*, have to be defined.

One can easily notice that a plethora of *asynchronous iterative* algorithms can be designed, even based on a simple iteration such as the Newton iteration. The problem arising is how to choose an algorithm. In any case, the advantage of *asynchronous iterative* algorithms is that processes are never blocked and the overheads due to the execution of the synchronization primitives are avoided. In fact, carefully selected *asynchronous iterative* algorithms can be proved to be very competitive to the best *synchronized iterative* ones. Research on the performance of *asynchronous iterative* algorithms is of great interest.

In [*KUNG76*] a particular consideration has been given to algorithms purely derived from the second of the previous strategies, which are called by him as *simple asynchronous iterative* algorithms. The main advantage of them is their general applicability; in other terms, they are not restricted to numerical iterative processes only, but they can be employed to speed-up any sequence of tasks, becoming particularly attractive when the task decomposition appears difficult. There are, however, some disadvantages, such as, the requirement for critical sections in the algorithms, and the fact that the speed-up obtained is quite limited if *fluctuations* in computation time, due to the system, are not large.

In a review of *synchronized* and *asynchronous* algorithms, since the former had the drawback that processes could be blocked and the latter that their analysis could be sometimes extremely difficult, a promising direction was to design iterative algorithms which were a compromise between these two types of algorithms, taking advantage of the special

features of individual iterations. These algorithms are called *semi-synchronized* (or *semi-asynchronous*) *iterative* algorithms.

The main characteristics of such an algorithm are that, it is 'loosely' synchronized so that processes are not expected to be blocked very often, thus reducing the *synchronized iterative* algorithms drawback; and, the synchronization guarantees that the iterations generated by the algorithm satisfy some desirable properties, thus reducing the *asynchronous iterative* algorithms drawback.

Finally, in [*KUNG76*] is introduced the special class of the *adaptive asynchronous* algorithms utilizing *global deques* (i.e. 'double-ended queues', see Knuth [*KNUT69*]), 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 an *adaptive* algorithm is obtained from the fact that the processes are able to adjust themselves during the computation, so that, they can all finish in about the same time. The concept of *adaptive* algorithms seems to be fundamental to the design of many efficient asynchronous algorithms.

In particular, the speed of an *asynchronous* algorithm may be improved if those processes not performing useful computations (a fact determined by examining the current contents of the global variables) can be interrupted promptly and if the extra overheads cost is not excessive.

In a review of all previously mentioned, we resume that *synchronized* algorithms should be utilized when *fluctuations* in process speed are small and when there are relatively few processes to be synchronized. To the contrary, when *fluctuations* in computation times are large, *asynchronous* algorithms are, in general, more efficient than *synchronized*

ones, since, processes never waste time in waiting for inputs, the algorithms can take advantage of running fast processes and they can be *adaptive* so that the processes can finish at about the same time. Furthermore, from the 'reliability' point of view, an *asynchronous* algorithm can be more reliable than a *synchronized* one, since, even if some processes are blocked forever, it can continue computing the solution of the problem, as long as, no blocking occurs in critical sections and there remain at least one active process.

To complete this paragraph and the concept of generally designing parallel, in particular, numerical algorithms, except their earlier discussed *complexity*, we should also briefly refer to their *numerical stability*, in order to conclude their quality criteria-picture. In the early days of parallel computing *complexity* was the prime aim. Only in the last few years has a growing interest developed for *numerical stability* or otherwise *insensitivity to round-off error*. More specifically, numerical algorithms on real computers produce results which, in general, contain a certain type of round-off error. This error can be divided into two components:

(i) *Inaccurate* input data will produce an error in the final result, even with an exact computation; and,

(ii) real-life computer systems produce round-off errors at *every step*, being *accumulated* up to the end of the computation.

An algorithm is defined as numerically *stable* if error (ii) does not exceed error (i).

To conclude, there are two methods of analyzing the numerical stability of an algorithm, the *Backward Error Analysis* and the *Forward Error Analysis* methods. The former one was proposed by Givens [*GIVE54*],

in 1954, and brilliantly utilized by Wilkinson [*WILK63*], in 1963, in the analysis of the rounding errors in algebraic processes. The question asked in backward error analysis is, 'What problem has one solved exactly and how far is this problem from the one set out to be solved?'. In other terms, this method mathematically transfers the round-off errors to the input data (variables). This may be contrasted with the usual forward error analysis where the question asked is, 'By how much does the computed answer differ from the true answer?'. In other terms, this method attempts to analyze the consequences of perturbations in input data (see Feilmeier [*FEIL82*]). The advantage of backward error analysis is that it is often easier to perform and the answers are often more useful.

II.B.3.1: Multiprocessors Performance Analysis Characteristics And Resource Provisions Of The '*NEPTUNE*' Parallel Processor System

Computer performance analysis, in general terms, is concerned with the analysis of computer systems *throughput* under various program loads. Through this analysis one can determine both, the kinds of load which can be processed efficiently by a specific computer system, as well as, any possible bottlenecks within the system that would cause the inefficient processing of other kinds of load. However, since a variety of resource amounts is provided by different computer systems, there exist some relative overheads from accessing these resources, depending upon the type of the resource, the programs (loads) varying demands for them (according to the way parallelism is introduced) and the design of the system itself.

The analysis of a *sequential* system performance can assist one

to determine the optimal provision of resources, such as I/O bandwidth and memory or processor speed. In terms of *parallel* processing, although all sequential processing features are of quite importance to the former, the effects of the new (specific to parallelism) resources must be taken into account.

More specifically, in this paragraph, by viewing performance as the interaction of resources demanded by programs and provided by a *Multiprocessor* system, we shall provide a performance prediction *framework* for parallel algorithms designed for such systems. The principle behind the *demand* and *supply* analysis of resources is that parallel computing involves the *sharing* of resources (i.e. processors, shared data structure, or a memory block), whose limited (finite cycle time) availability forces program demands to compete to 'own' them.

More analytically, this competition or contention has *three* consequences:

(i) The theoretical limit to the number of demands that can be satisfied constrains the maximum system or program performance;

(ii) a mechanism (*allocation* algorithm) is necessary to resolve amongst competing demands, which itself imposes an overhead on resource access even in the absence of contention; and,

(iii) an implication of the (i) factor is that some of the competing demands will be forced to 'wait' for the specific resource to become available.

The last two performance degrading factors are called respectively the *static* and *dynamic* costs of shared resource access.

Consequently, by analyzing these system properties (i.e. resources availability and allocation algorithm) under various theoretical demand

patterns and by characterizing a program demands, these two pictures can be superimposed to yield the performance of a particular algorithm on a particular piece of hardware. This analysis will be substantiated by a brief analysis of the resources provided by the *NEPTUNE* parallel processor system.

To review, the main principle of parallel computing is that it requires three primary resources: *Multiple* processors, *communication* for data sharing (even if this is only as much as required to start processing in the first instance), and *synchronization*† to allow unique data modification (although it is possible to construct asynchronous programs, it is not the norm).

Two alternative overall performance measures of a program, reflecting respectively the differing, algorithm and system designer, aspects, are the *ratio* of its *time-complexity* on a p-linked processors system, to the *time-complexity* on a single processor; and, the *percentage* of the potential output of p independent processors, linked together and cooperating on a number of programs.

In the following, we shall discuss, more analytically, some of the inherent limitations on the performance of a p-processor parallel system due to some either apparent (or not) types of overhead, associated with the *Multiprocessor* execution but not with that on a uniprocessor.

Primarily, it is obvious that a p (all identical)-processor system cannot complete a task more than p times faster than a single processor and therefore the speed-up factor is limited by the number and power of the processing elements. However, it is also limited by other factors introduced by the 'cooperation' required amongst all the processors

†*It is not the 'synchronization primitive' alone that is the resource, but it and the structure it protects, and if there are several logically independent structures each should have its own measure.*

performing on a common task; in fact, conceptually there is a shared database which may be required to be changed some times dynamically during processing.

Also, the fact that a task must be subdivided into p (or more) individual subtasks, either before the processing can start or during processing, causes an additional overhead explicitly associated with a *Multiprocessor*.

The above task subdivision may give rise to a further three possible types of overhead, unique to the nature of a *Multiprocessor* system:

(i) In the case that less than p subtasks are available, at any time, to be allocated for execution, then some of the processors must remain *idle*. This idle status length can be estimated (even predicted), if the processors speed is known, by examining the *computational-complexity* and the number of subtasks;

(ii) an *organizational* overhead, associated with ensuring the proper sequencing of the execution of subtasks, occurs when one subtask generates results required as input to another subtask; in this case, the latter subtask has to 'wait' until the former subtask produces the results. In a similar way, an additional overhead can incur when one subtask must 'wait' because the information is not available. The difference with the former *organizational* overhead is that the latter can occur in the middle of subtask execution, thus being equivalent to waiting for a resource; and,

(iii) in the case that the *shared* database can be simultaneously accessed, at any time, by a lesser number of processors than the system comprises, then, an overhead incurs associated with

the checking of the number of simultaneous accesses not to exceed the limit, which also causes a waste of time if processors must 'wait' to gain access.

In conclusion, two distinct sources of overheads can be distinguished, those due to the design of software and hardware 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, checking by hardware and software for contention when accessing the shared database, as well as, checking for correct sequencing; these are all called *static* overheads, since, once the number of processors, the methods of communication, synchronization and task allocation, for the algorithm to be processed, are decided, then the number of subtasks, synchronizations and accesses are all properties of the algorithm itself.

The other source of overheads concerns the so-called *dynamic* overheads which depend 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 occasions.

From the measurement point of view, the *static* overheads can be determined. In particular, if subtasks are created and allocated to processors 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; in a similar way, by knowing the cost of one access or synchronization, the relative overhead is estimated.

The same way does not apply though for the estimation of *dynamic* overheads, where, usually, a statistical estimate can be made depending

on the occurrence and duration of events (subtask creation, resource demands, synchronization) model.

As it was mentioned earlier, the performance of a *Multiprocessor* can be expressed by the usual speed-up factor, but also, it can be expressed in terms of the time not utilized productively, namely, the 'Wasted-time' (W). For the latter case, the formula is:

$$W = p * T_p^{\dagger} - T_s \; ; \qquad (II.B.3.1:1)$$

the wasted-time must be equal to the 'sum' of the *static* and *dynamic* overheads.

Consequently, by ignoring the algorithm design time, it would be:

$$W = \underbrace{tA + \sum_j q_j B_j + \sum_k r_k C_k}_{\textit{static}\ \text{overheads}} + \underbrace{\sum_{i=1}^{p} (W_i + \sum_j X_{ij})}_{\textit{dynamic}\ \text{overheads}} \; , \qquad (II.B.3.1:2)$$

where, t : is the number of subtasks

A : is the creation and allocation overhead for a task

B_j : is the overhead associated with an access to the j^{th} type of resource

q_j : is the number of accesses to this resource

C_k : is the overhead associated with the k^{th} synchronization method

r_k : is the number of such synchronizations

W_i : is the time the i^{th} processor is 'idle' waiting for the allocation of subtasks

X_{ij} : is the time the i^{th} processor waits for access to the j^{th} resource.

Both measures (i.e. *Speed-up* and *Wasted-time*) are interrelated,

---

†*For a 'p'-processor system.*

since it is clear that either all processors complete processing together, or some processors take longer than others; consequently,

$$T_p \geq \frac{T_s + W}{p} \iff S_p \leq \frac{pT_s}{T_s+W}, \qquad (II.B.3.1:3)$$

and the maximum speed-up factor, for a given algorithm, can be determined by assuming the *dynamic* overhead to be zero†.

Finally, and despite the requirement of an exact pattern for the shared resources demands to determine in detail the waiting times, it is possible to estimate the bounds on the maximum number of processors, that can be utilized efficiently on an algorithm, from the average utilization figures for each resource, for each subtask. In conclusion, since different hardware and software gives rise to different *static* costs, therefore, by knowing the specific overheads of the system to be utilized, algorithms may be accordingly designed to minimize these costs.

In the remainder of this paragraph, we shall discuss the actual overheads observed on a system with shared memory, the *NEPTUNE* parallel processor system at Loughborough University. The ability of this system to provide resources statically (i.e. when requests for resources do not contend each other) has been summarized in *Table (II.B.3.1-t1)*.

However, before we proceed with this, we must mention that a communication based system consisting of individual machines, linked together by a communications subsystem, involves different *static* and *dynamic* overheads. More specifically, the time a machine spends to establish local copies of the required information from the shared database, the time required for the messages exchange amongst machines and the updating time of the local databases to incorporate these

†*If subtasks are available to processors throughout the tasks execution period and no resource is ever busy when requested, then it is possible for the 'dynamic cost' to be zero.*

messages, consist the prime *static* costs for such a computer complex. In particular for the performance limitations due to resource demands, at the hardware level[†], they incur only from the loading on the communication system, the only shared resource at this level.

To return to the *NEPTUNE* parallel processor system, specifically for the three necessary for parallel computing resources, mentioned earlier, and from the processors aspect, the software controlling the processors scheduling to processes counts the number of processes run by each processor; then, the *static* cost of parallel control is estimated as the product of this number and the cost of scheduling given in *Table (II.B.3.1-t1)*. Also, the 'idle' processor time, because there are not available processes to be allocated to them, can be estimated by this software.

On the other hand, the generation of the parallel processes and their allocation to processors is done *dynamically* and this scheduling is achieved utilizing a shared list of processes protected by *mutual exclusion*. In *Table (II.B.3.1-t1)* one can see the average time this resource is 'blocked' to other processors, whilst software can estimate the dynamic loss of performance due to this contention for the resource.

From the *shared* memory point of view, the *static* cost of each processor to access it, arising out of the hardware multiplexing of more than one processor into a single memory block, is also given in *Table (II.B.3.1-t1)*. Contention increases the *static* overhead and is widely recognized to be a function of the number of contending processors and the temporal pattern of access to the block of shared memory; however these losses are markedly smaller, than the contention

[†] *At the software level other logically shared resources can exist e.g. the local database of a machine may be required to be accessed by subtasks running on other machines.*

Resource \ Processor	P_0	P_1	P_2	P_3
Relative Speeds†	1.000	1.037	1.006	0.978
Floating Point*	~700	~700	~700	~700
Integer*	~ 20	~20	~20	~20
Local Memory Access*	~0.6	~0.6	~0.6	~0.6
Shared Memory Access*	~1.41	~1.12	~1.31	~1.32
Mutual Exclusion Mechanism*	~400	~400	~400	~400
(blocked)*	~200	~200	~200	~200
Parallel Path Mechanism*	~800	~800	~800	~800
(blocked)*	~400	~400	~400	~400

- † excluding access to shared memory
- * times in microseconds

Table II.B.3.1-t1‡: Resource Provisions of the *'NEPTUNE'* System.

losses arising from mutual exclusion, due to normally more regular (or synchronous) access pattern to shared memory.

Finally, the fact that a high level software technique has been adopted, for the *mutual exclusion* to overcome hardware existing inadequacies, has increased its cost significantly. Thus, although hardware 'test' and 'set' operations are available (with *XPFCL* command - see *Appendix C-II/par.-II.A.3-i*), as a basis for claiming resources, they cannot be accessed directly from the user program. Furthermore, these operations 'block' the shared memory for 10μs and, under high load conditions, the resultant waiting by other processors, for shared memory access, could sufficiently exceed the memory access time-out, thus

‡*Most of the timings in the Table, due to amendments performed on the system, have been altered (see Appendix C-II/par.-II.B.3.1).*

'crashing' the system (see *XPFCLD* command-*Appendix C-II/par.-II.A.3-ii*).

The routines ensuring *mutual exclusion* to shared data structures, count the number of times each processor accesses each distinct data structure; 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 (II.B.3.1-t1)*. In addition, these routines also can estimate the 'waiting' time, due to the contention for each of these resources from each of the processors. The control routines themselves add a small amount of time, to the time for which the relevant resource is blocked to other demands (see *Table (II.B.3.1-t1)*).

To conclude the discussion, we must clarify a significant factor 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 processes 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, that can be effectively utilized in a parallel program, is limited independently by each shared resource according to: *Max. no. of processors = 1/(demand rate * 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.

CHAPTER III: FIFTH GENERATION KNOWLEDGE-BASED AND VLSI CHIP-EMBODIED INFORMATION FLOW COMPUTER SYSTEMS

SECTION A: COMPUTATIONAL MODELS AND 'KNOWLEDGE INFORMATION PROCESSING SYSTEMS' - *KIPS*

III.A.1: INTRODUCTION

III.A.2: APPLIED SCHEMAS FOR DESCRIBING PARALLELISM IN COMPUTER SYSTEMS

III.A.3: OBJECTIVES OF 'FIFTH GENERATION COMPUTER SYSTEMS' - *FGCS* AND NOVEL DECENTRALIZED MACHINES AS THEIR POTENTIAL ARCHITECTURAL BASIS

III.A.2.1: ORIGINATION AND MODELLING POTENTIAL OF '*PETRI NETS*'

III.A.2.2: EXTENSIONS, SUBCLASSES AND RELATED MODELS TO '*PETRI NETS*'

III.A.2.1.1: THE STRUCTURE, MODELLING PROPERTIES AND EXECUTION RULES OF '*PETRI NETS*'

III.A.2.1.2: '*PETRI NETS*' ANALYSIS APPROACHES, PROGRAMMING CONSTRUCTS REPRESENTATION, AND FORMAL LANGUAGES

SECTION B: EMBEDDING INFORMATION FLOW SCHEMES ON GRIDS AND IN CHIP AREA AND TIME

III.B.1: THE IMPACT OF THE TECHNOLOGICAL INNOVATION ON FUTURE ARCHITECTURES - THE *VLSI* CHALLENGE

III.B.2: FUNDAMENTAL ARCHITECTURAL CONCEPTS IN DESIGNING SPECIAL-PURPOSE *VLSI* COMPUTING STRUCTURES

III.B.2.1: THE FUNDAMENTAL PRINCIPLE, CRITERIA AND ADVANTAGES OF '*SYSTOLIC*' ARCHITECTURES

III.B.2.2: A COMPATIBILITY TAXONOMY IN THE SPACE OF '*SYSTOLIC*' COMPUTATIONS AND *VLSI* STRUCTURES

III.A.3.1: ORIGINATION, FUNDAMENTAL HARDWARE AND SOFTWARE PRINCIPLES, CHARACTERISTICS OF THE '*DATA FLOW*' MACHINE ARCHITECTURES

III.A.3.2: A GENERAL SPECIFICATION OF RESEARCH SUBJECTS AND CHARACTERISTICS FOR A *FGCS* '*DATA BASE*' MACHINE ARCHITECTURE

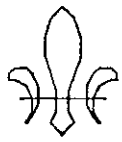
III.A.3.1.1: PROTOTYPE '*DATA FLOW*' MACHINE ARCHITECTURES, PROGRAMMING LANGUAGES, AND FURTHER DESIGN ALTERNATIVES

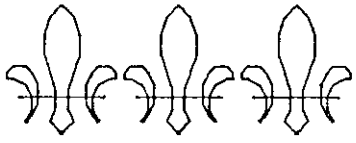
Chapter III

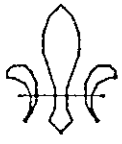
Fifth Generation Knowledge-Based And VLSI Chip-Embodied Information Flow Computer Systems

SECTION Å

COMPUTATIONAL MODELS AND 'KNOWLEDGE INFORMATION PROCESSING SYSTEMS' - *KIPS*







Louis XVI : *'C'est une grande révolte.'*

'It is a big revolt.'

La Rochefoucauld-Liancourt : *'Non, Sire,*

c'est une grande révolution.'

'No, Sir, it is a big revolution.'

Duc de la Rochefoucauld-Liancourt

1747–1827

F. Dreyfus,

La Rochefoucauld-Liancourt

1903, Ch. ii, Sect. iii

III.A.1: INTRODUCTION

Today's requirements for high-speed computer systems became apparent in an ever increasing number of applications. In fact, the success of *CRAY*-like type of vector processors has shown the requirement for very fast computer architectures to perform complex scientific computations. However, almost all of the various types of multiple processor systems, mentioned in previous *Chapters*, do not seem to fit the performance goals that scientists would like to be met by the next 'Fifth Generation Computer Systems' - *FGCS*, whatever the technological advancements would be in the future.

The *Speed-up* and the better *performance* in general, up to now, have been principally achieved by architectural improvements and the development of compilers to map programs onto any particular computer architecture.

With the advent of 'Very Large Scale Integration' - *VLSI*, the design of high-performance digital computer systems changes from the realization of an algorithm on a given architecture, fabricated in a fixed technology, to an integrated process. *FGCS*, by handling these

rapid developments in microelectronics, will represent a unification of the research into *VLSI* processors and into distributed processing, aiming to solve very complex problems with a very high level of performance.

Each computer system will consist of a network of computing elements supporting an individual application or requirement; these computing elements, through *VLSI* technology, will provide either a general-purpose or a special-purpose function, ranging in power from a *miniature* microcomputer to a *mainframe* computer. If a large number of computing elements are to work together in a *FGCS* or otherwise *decentralized* computing system, then it is necessary to have a harmonious set of architectural principles which the program and computer organizations will support. This is true whether the decentralized computer system is composed of miniature computing elements, within a single board or even chip, or consists of geographically distributed mainframe computers.

However, if the design of such computer systems is one aspect of the problem, the successful design of these systems is another. In order for the latter aspect to be accomplished, systems must be seen to be the result of integrating together several different *constraint domains*, each of which has its own characteristic structures; these are, *technology*, *algorithm*, and *architecture* in the first place and, without diminishing their importance, *data structure* and *programming language* in the second place, the mutual interaction of which will establish a balanced design satisfying the system goals. It can be certainly claimed that the best designs will be obtained when these structures 'match' or interrelate in a complementary way, each supporting the other.

All of these constraint domains are of course *knowledge sources*,

each with a separate substantial internal theory; but, an overlap with the other domains may occur, a fact which implies that the overall system designer must be well aware of the design freedom and constraints at each level, as well as, the consequences of design decisions at any given level on all of the other interacting levels. However, it should be pointed out that the first three knowledge sources are more complex ones, with a non-trivial overlap.

In many problem domains, from the algorithmic and architectural representation aspects, we are often given an initial algorithm and architecture together. However, unfortunately, this fact confounds the *performance* of the algorithm, as we have seen it[†], with what the algorithm really does, or, otherwise, its *competence*. This is mainly because the algorithm is expressed in terms of an architecture, which up to now was usually a von Neumann single-sequence computer architecture; but, this fact tends to obscure the potential parallelism, and represents the algorithm at only one performance level. It would be more preferable to have a fundamental means to represent separately the performance and competence of an algorithm.

However, it is possible to introduce competence-constrained performance transformations which, while guaranteeing the competence preservation, allow a methodic performance manipulation; for example, by utilizing some techniques depending on basic ideas of conflict resolution, a system specified by a hardware design language scheme can be transformed to any other theoretically possible performance.

In particular for the technological domain, it is far ahead than

---

[†] *Namely, how the algorithm is executed.*

the rest of the field and the recent advances in integrated circuit technology have led to a rapid expansion of the research into highly parallel and specialized computer architectures; for an excellent *state-of-the-art* survey, the reader should refer to the *January 1982* issue of *IEEE* Computer Magazine.

The realization of a particular architecture, in a selected technology, is done hierarchically in a way that displays the available useful parallelism; this means that module interconnections must be made explicit (topologically and in area utilization), a fact which induces close coupling between architecture and technology.

Today, packing densities of tens of thousands of transistors per chip (*LSI*) are possible and into the near future factors of *10* to *100* times this amount (*VLSI*) are predicted. Integrated circuits are thought of as *2*-dimensional arrays of devices now; however, with the introduction of stacked epitaxial silicon layers, *3*-dimensional structures are possible. Furthermore, the metal interconnect and contact technologies have very strong effects on the architecture and its performance in both, space and time. Consequently, it is necessary to carry out research programmes to effectively exploit the potential, being offered by *VLSI* technology, to construct highly sophisticated 'plug-in' processors to form a parallel system.

Finally, of similar significance to the increase in packing density is the rapid development of sophisticated *CAD* tools, which will drastically cut down the design time and make the technology available to a much wider user community.

To conclude, the requirements of parallel architectures for *VLSI* have been discussed by a plethora of authors (see Kung [*KUNG82*], Seitz [*SEIT82*]) and will be briefly analyzed in *Section B* of the present *Chapter*.

III.A.2: Applied Schemas For Describing Parallelism In Computer Systems

An overview of our brief and selective *top to bottom* descriptive approach to the general topic of parallel processing (i.e. classification of designs, global architectural structures, processing units description, applications, programming languages, design and particular analysis of parallel algorithms), which was introduced in the previous *Chapters*, can reveal that these descriptions have to be considered as *'static'*, in the sense that they simply delineate either, an overall computer structure, or, depict program instructions and their dictating rules for execution. Also, the way information flows through *PMS*[†] primitives is of course graphically indicated by buses and, within particular systems, by programming languages' control statements. However, the *'dynamic'* sequences of events, required to perform particular actions, are not represented in as clear a fashion as one would like.

This is not a difficulty inherent in Computer Architectures. There are very few algorithmic processes, amenable to a certain notation, reflecting the sequencing of imminent executions. *Regular expressions*, drawn from formal language theory, are the most evident examples of such a case.

In particular for Software Engineering, *flowcharts*, without any

---

[†] *A notation due to Bell and Newell [BELL71], comprising the following 'seven' primitive components: 1) 'Memory~M' (stores information over time without modifying its contents or format), 2) 'Link~L' (connects other components and transfers unaltered information), 3) 'Switch~S' (builds links or transfer paths of control and data), 4) 'Control~K' (evokes the other PMS components), 5) 'Data-Operation~D' (alters units of information, like an ALU), 6) 'Transducer~T' (represents the communications with the external environment), and 7) 'Processor~P' (not a purely primitive component, comprising all the previous six components, being able to interpret a stored program).*

gain in formalism, depict, in a very similar manner to the source programming language, the executional procedure. However, although flowcharts assist in the understanding of a process, they are restricted to the modelling of the flow of control, while the representation of data structures and data flow is generally non-existent.

In Systems Architecture several factors may increase the complexity of the representation, especially when one wishes the overall descriptions to include means for analysis and evaluation, i.e. provide a design apparatus. In general, such an apparatus should present satisfactory the *flow* of control, the *structures* (hardware and software 'data structures') in which information is stored and utilized, the creation and deletion of processes, and the allocation and occupancy of resources during the processes' lifetimes.

Currently, there is not such a completely successful modelling media and this is an area of intense investigation which has a variety of rather different sounding *applied schemas* (or *models*) to show; these schemas are applied explicitly and consequently they could, somehow, be cited in continuance to the *explicit* approach to parallelism discussed in the previous *Chapter*.

The author though, urged by futuristic concepts (see *par.-I.B.6*), has placed them here, since their modelling and decision power, so far, offers a certain clearness and cleanness; this permits a simple, natural and conceptually closer to the next generation, representation of computer systems, a fact justifying their gained increasing acceptance and utilization.

Some of these schemas, depending on their intended application, but

without being oblivious to the underlying computer architecture and its software, aim mostly at *formal* descriptions, hence suitable for correctness analysis (see Miller [*MILL73*]); while others are more attuned to the description of large systems, less formal and more akin to parallel flowcharts (see Baer and Jensen [*BAER77*]).

Most of the notational means, that are utilized to describe, model and study computer systems, are probably best thought of as *graphs*. Pipelined or Array computer architectures are naturally described utilizing graphs, in a similar way that programs are represented as flowcharts, flowgraphs, or other graph-like structures. The branch of mathematics known as *Graph Theory* offers useful terminology and notation to describe computer systems; therefore, a brief retrospection of some, conceptually required for the subsequent paragraphs, preliminary definitions about *graphs* is absolutely imposed.

- *Preliminary Graph Definitions*

d_1: A *connected graph* is simply a set of 'nodes' connected by 'links' (each link joined to two nodes, called its *ends* or *parts*). Terminology, though, is not standard and hence, the reader may find in the literature 'vertices or points' and 'edges or lines' as synonyms for 'nodes' and 'links', respectively.

d_2: The number of links joined to a node defines its *degree*; in the case that all nodes are of the same degree, then the graph is called *regular*.

d_3: When there are several paths between a pair of nodes, these form *cycles*, since they give paths from a node back to itself; a graph free of cycles consists of a *tree*.

$\underline{d_4}$: When a link exists between every pair of nodes then the graph is called *complete*. A graph which can be partitioned into two subsets of nodes, so that each node is joined to every node in the other subset, is called a *complete bipartite* graph.

$\underline{d_5}$: A graph having one-way links (rather than undirected two-way links) is called a *directed graph* or *digraph*.

The term *network*, in this Thesis and elsewhere in the literature, has been utilized more generally in order to designate computer architectures with several processors and memories, and as a synonym for graph; however, in the context of *network flow* problems it is defined as a *directed graph* whose links are assigned non-negative integer values (designating, e.g. channel capacity or power), with two usually disjoint subsets of nodes, the *source*[†] and the *sink*[‡].

Our bias, in the related paragraphs which follow, is towards a less formal pragmatic approach requiring as a first quality of the schemas that they be descriptive of the control and information flows they represent. This fact leads us towards *graph models* to which we can assign various levels of interpretation.

Amongst the modelling media, which have been utilized to represent concurrent activities, *Petri Nets* (in the first place) and their extensions and subclasses have been quite popular. Their modelling of control flow, compared to flowcharts, is performed in a richer and more formal way and when supplemented by some data representation they can be of specific interest to Systems Architects.

[†]*Where, e.g. raw material or initial information is input.*

[‡]*Where, e.g. finished products or solutions to problems are output.*

Because of *Petri Nets* powerful general potential, a fact *a priori* establishing them as the 'Kernel' of all modelling media, and since the literature on them is quite abundant, the author will restrict himself to a particular, but brief and informal, introduction.

III.A.2.1: ORIGINATION AND MODELLING POTENTIAL OF '*PETRI NETS*'

In many sciences, a *phenomenon* is studied by examining not the actual phenomenon itself, but rather a *model* of the phenomenon. To successfully utilize the modelling approach, however, requires a knowledge of both, the modelled phenomena and the modelling techniques. Then, by the appropriate manipulation of the model, it is gracefully expected that new knowledge about the phenomenon under examination, as well as the model itself, will be obtained without any cost, inconvenience, or danger that would emanate from the manipulation of the real phenomenon itself.

Most modelling makes use of mathematics. In many physical phenomena their significant characteristics can be illustrated numerically, with equations or inequalities describing the holding relations amongst them. For example, the Differential Calculus was developed in direct response to the requirement for a means to model continuously altering properties, such as, position, velocity or acceleration in Physics.

Petri Nets have been utilized as rigorous models for modelling the dynamics of a system, as *Turing Machines*† (see Turing [*TURI36*]) have

---

†*Suitably powerful 'finite-state automatons', with 'potentially infinite' memory, capable of doing anything that any other computer might conceivably do - given enough time.*

been for sequential computer organizations. More specifically, *Petri Nets* are abstract and formal modelling media, with properties, concepts and techniques especially devised in a search for natural, simple and powerful methods for describing and analyzing the information and control flow for the category of *discrete-event* systems. These systems may exhibit asynchronous and concurrent activities, more likely under certain constraints on the concurrence, precedence or frequency of them. In this aspect, at any given time, certain conditions will hold, a fact which causes the occurrence of certain events. In accordance, this may lead to a change of the system status, causing some of the previous conditions to cease holding and others to begin then to hold.

A brief historical retrospection reveals that, the theory of *Petri Nets* originated in the early (1962) work of Carl Adam Petri, in Germany, who in his thesis [*PETR62*] developed a new model of information flow in systems. This model was based on the concepts of asynchronous and concurrent operation by the parts of a system and the realization that relationships between the parts could be illustrated by a *graph* or *net*.

Later on, Petri's ideas came to the attention of a group of researchers, led by Anatol Holt, at Applied Data Research, Inc., working on the Information System Theory Project (see Holt, et al [*HOLT68*]). This group developed the theory of *'Systemics'* (see Holt and Commoner [*HOLT70*]), concerned with the systems representation, analysis and behaviour. It was this specific work which provided the early theory, notation, and representation of *Petri Nets*, showing the way they could be applied to the modelling and analysis of systems of concurrent processes.

Petri, since then, has expanded upon his original theory, carrying out work on the basic concepts of information flow and the structure of concurrent systems. This has resulted in a form of general systems theory, called *Net Theory* (see [*PETR73*], [*PETR75*]) which is related to Topology.

In contrast to the work of Petri, Holt and other European researchers, which emphasizes the fundamental concepts of systems, evolving into a more general and abstract theory, the work carried out in some American research centers (e.g. at MIT) concentrates on those mathematical aspects of *Petri Nets* closely related to *Automata Theory*. According to this approach, systems are being modelled as *Petri Nets*, the convenient manipulation of which assists in deriving the properties of the modelled systems.

This *mechanistic* approach is quite different in orientation from the more *philosophical* approaches of Petri and Holt, and requires the development of techniques for analyzing *Petri Nets* in order to answer questions arising, like: 'What *markings*[†] are *reachable*[†] in a given *Petri Net*?','What sequences of *transition*[†] *firings*[†] are possible?',etc.

Finally, in a reviewing and hierarchically delineate manner, from the modelling potential aspect, there are certain areas in which *Petri Nets* would seem to be the perfect media to implement: Those, in specific, areas in which events occur asynchronously and independently. Large, powerful computer systems often utilize asynchronous parallel activities, in an attempt to achieve optimal parallelism and hence increase effective processing speed, while of course being *determinate*, i.e., still

[†] *This notational concept will be introduced in the following paragraph.*

produce correct results; consequently, *Petri Nets* can be naturally utilized in the modelling of *'Hardware'*. In particular, *Petri Nets* have been associated with the description of general modular asynchronous systems (see Dennis [*DENN70*], Patil [*PATI72*]), and macromodules (see Clark [*CLAR67*]). In addition, at Honeywell, *Petri Nets* have been utilized for investigating the fault-tolerant properties of designs (see Jack [*JACK76*]).

At the *Software Engineering* design level, Operating Systems (e.g. resources allocation, deadlock situations, processes coordination, etc.), Compilers, and Distributed Database update algorithms, are some of the applications whose control flows have been modelled with (extensions of) *Petri Nets*.

To conclude, a variety of other research areas have been mentioned as possible implementation subjects of *Petri Nets* schemas, including Queueing Networks, Traffic Control, Distributed Computer Systems, Legal Systems (see Meldman and Holt [*MELD71*]), Proofs in Mathematics (see Genrich [*GENR75*]), and even Brain Modelling.†

III.A.2.1.1: The Structure, Modelling Properties And Execution Rules Of *'PETRI NETS'*

The development of an appropriate theory has motivated most of the research on *Petri Nets*, since the efficient utilization of them requires a careful understanding of their structural nature and

---

†*The neuroscientist W. McCulloch and the mathematician W. Pitts, in 1943, first proposed an idealized 'neuron-net' calculus for modelling the brain's computations [MCCU43].*

facilities; in fact, for a *Petri Net* to accurately model a system, it must ensure that all events sequences, possible to be met in real life, have been included. Our introduction, herein, to their basic concepts, will be presented briefly and less rigorously defined and formalized, than they appear in the literature. The interested reader in a more formal treatment should consult the provided references.

The pictorial representation of a *Petri Net* by a graph is a common practice in this research field. In fact and according to the preliminary graph definitions of (*par.-III.A.2*), a *Petri Net* is a *directed bipartite graph* containing two types of nodes: 'Circles' (called *'places'*) corresponding to *conditions* which may hold in the system, and 'bars' (called *'transitions'*) representing the *events* which may occur. If a 'link' (or *'arc'*) is directed from node i to node j (either, from a place to a transition, or vice versa), then i is an 'input' to j, which in turn is an output of i; for example, in *Figure (III.A.2.1.1-f1)*, which depicts a simple graph representation of a *Petri Net*, place p_1 is an input to transition t_2, while p_2 and p_3 are outputs of transition t_2.

A major feature of *Petri Nets* is their *asynchronous* nature, which in other terms means that there is no inherent measure of time, or the flow of time in the net. Since in real life events take variable amounts of time, *Petri Net* models should reflect this variability by not depending upon a notion of time to control the occurrence of events; therefore, a *Petri Net* structure, itself must contain all necessary information to define the possible sequences of events of a modelled system.

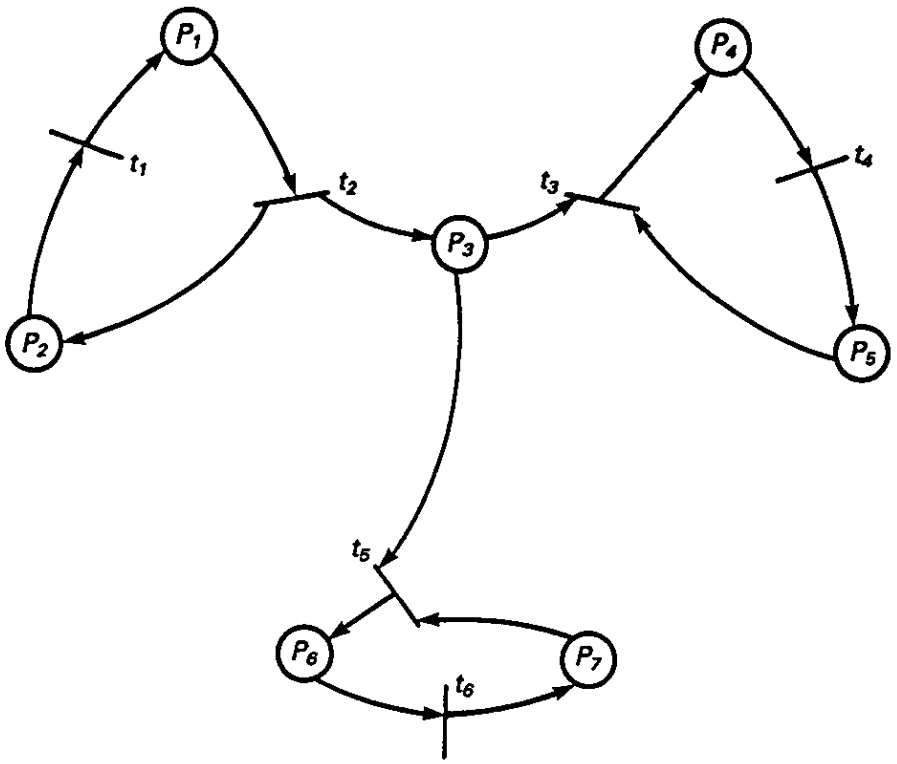


Figure III.A.2.1.1-f1: A Simple Graph Representation of a *'Petri Net'*.

A *Petri Net* graph, in general, models the *static* properties of a system, much as a flowchart represents the static properties of a computer program. In addition to the *static* properties represented by the graph, a *Petri Net* has *dynamic* properties resulting from its execution. The execution of a *Petri Net* is controlled by the position

and movement of 'markers' (called *'tokens'*), indicated by black dots in the *places* of the net.

In an informal definition-like manner, a *Petri Net* with *tokens* is a *'marked Petri Net'*. The distribution of *tokens* in such a net, called its *'marking'*[†], defines the *state* of the net and the set of all its markings forms the *state space* of the *Petri Net*. *Tokens* are moved by the *'firing'* of the net *transitions*. A *transition* is *'enabled'* to fire, when all of its input *places* are full. A *transition* is *'potentially firable'*, if there exists a sequence of *transition* firings that enables it. In different markings, different *transitions* may be enabled. However, the lack of uniqueness in the event occurrence leads to a *non-determinism* in execution; this means that if at any time more than one *transition* is enabled, then any of them may fire, the choice being imposed randomly, or by unmodelled forces. The result of the *transition* firing is to remove one *token* from each of its input *places* and to put one *token* on each of its output *places*. No net change occurs, if a *place* is neither an input nor an output, or is both[‡] an input and an output.

The *non-deterministic* manner of execution, albeit it introduces some modelling advantages, certainly causes a considerable complexity into the analysis of *Petri Nets*. To reduce this complexity, the firing of a *transition* is considered to be *instantaneous*, i.e. take zero time; then, since time is a 'continuous' variable, the probability of any two or more events being executed simultaneously is zero, which simply

---

† *A marking 'M' is notationally presented as a vector 'M'=($\mu_1,\mu_2,\ldots,\mu_n$), where each component μ_i represents the number of tokens in the 'i^{th}-place' of a 'Petri Net' with 'n' 'places' in total.*

‡ *In this case it is necessary for a 'token' to be in the input 'place', albeit no change in marking occurs for this 'place'.*

means that two *transitions* cannot fire simultaneously. The events being modelled that way are considered as *primitive* events; in the case of no primitive ones (i.e. events requiring non-zero time), they are decomposed into a 'beginning' and an 'ending', which are *instantaneous* events, plus the *non-instantaneous* occurrence. In the case that marking is such that some enabled *transitions* are dependent on each other (i.e. share a common input *place*), then they are said to be in *'conflict'*. These *concurrency* and *conflict* concepts, which are basic to an understanding of *Petri Nets*, are illustrated in *Figures (III.A. 2.1.1-f2,f3)*.

The firing of *transitions* may change the net marking, and it, i.e. the execution sequence, may continue as long as there exists an enabled *transition*. It should be noted that a *token* in a *place* can be utilized in the firing of only one *transition*, and a *Petri Net* is considered *'safe'*, if a *place* cannot hold more than one *token* at any time. In addition, a *Petri Net* is called *'conservative'*, if the number of tokens in the net is conserved.

A marking M' is *'immediately reachable'* from marking M , if the firing of some enabled *transitions* in the marking M results in the marking M'. A marking M' is *'reachable'* from M, if it is immediately reachable from M, or is reachable from any marking which is immediately reachable from M. The set of all reachable markings from M is denoted as $r(M)$.

A *transition* t is *'live'* for a marking M, if for all markings $M'' \in r(M)$ there exists an execution sequence which reaches M'' where t can fire. A *Petri Net* is called *'live'*, if all its *transitions* are live.

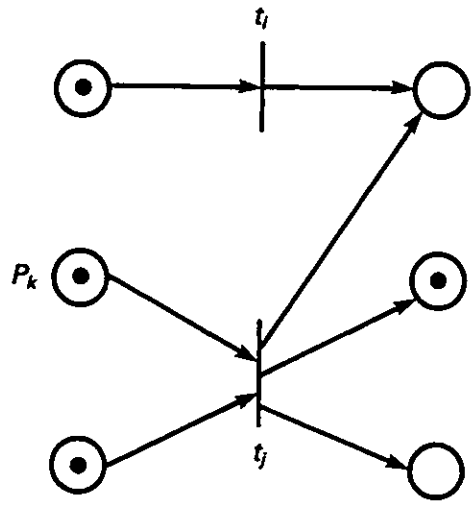


Figure III.A.2.1.1-f2: The Modelling of either *Firing* Order *'Concurrent'* Events.

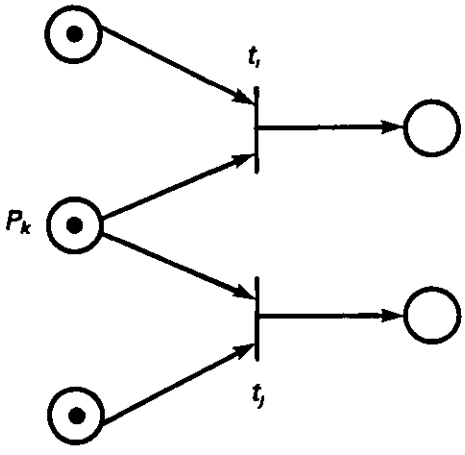


Figure III.A.2.1.1-f3: The *Firing* of either of the *Transitions* t_i, t_j Disables the other (i.e. in *'Conflict'*).

This *liveness* property is related to the absence of 'deadlock', that of *safety* to boundedness in the utilization of resources, and *conflicts* are utilized to model synchronization constraints, as well as predicates.

Another important aspect of *Petri Nets* is that they are *uninterpreted* models; in other terms, no meaning is attached to the *places* and *transitions* in this sort of nets dealing only with the abstract properties inherent in their structure.

Finally, a valuable feature of *Petri Nets* is their ability to model a system *hierarchically*. An entire net may be replaced by a single *place* or *transition* for at a more abstract level modelling (*'abstraction'*), or *places* and *transitions* may be replaced by subnets to provide more detailed modelling (*'refinement'*).

To conclude, the reader must realize that terminology, notation and emphasis have varied widely in research on this subject, a problem principally caused by *Petri Nets* power and the resultant diversity of applications; however, as the author mentioned at the beginning of this paragraph, the interested reader may refer to the original works, proofs and details, guided from the given Bibliography in J.L. Peterson's paper (see [*PETE77*]).

III.A.2.1.2: '*PETRI NETS*' Analysis Approaches, Programming Constructs Representation, And Formal Languages

In a historical-like retrospection, *Petri Nets*, with their uniform and simple execution rules, were originally oriented to play just a descriptive (and sometimes designing) role, presenting systems of

asynchronous concurrent processes in terms of simple and natural concepts. However, after a short time, it became obvious that another significant utilization of *Petri Nets* could be in the *analysis* of the systems description for the location of desirable or not desirable properties.

In actual fact, the first task in developing analysis techniques is to define the sorts of questions that the analysis procedures are to answer, as well as the properties to be studied. It is obvious that these analysis techniques should, rightfully, be biased towards the solution of the most commonly (required to be solved) problems, rather than problem areas of academic only interest.

There is a variety of interesting questions that might be studied with *Petri Nets*. Therefore, it is of high importance for general techniques to be developed, which are capable of answering new questions, since, even the questions designers themselves wish to inquire about their designs depend on the projected utilization of them. However, some of the analytic questions that one would, probably, like to inquire about a *Petri Net* are quite difficult; hence, restricted subclasses of *Petri Nets*[†] have been defined to achieve an easier analysis in specific situations.

We must emphasize on the fact that many questions can often be reduced to the so-called *'reachability problem'*[‡], which is of high significance to the analysis of *Petri Nets*. This problem can be considered as a special case of the *'set reachability problem'*, which

---

[†] *We shall discuss them in (par.-III.A.2.2).*

[‡] *The 'reachability problem' is: 'Given a marked 'Petri Net' (with marking 'M') and a marking 'M'', is 'M'' reachable from 'M'?'.*

is to determine if a given set of markings is a subset of the reachability set $r(M)$ of a marked *Petri Net*.

Despite the fact that for the analysis of *Petri Nets* several approaches have been considered, one basic technique is utilized by almost all researchers in this area; the aim of this technique is the determination of a *finite* representation for the reachability set of a *Petri Net*, since most of the net properties are based on properties of this set. The representation of this set is best known as the *'reachability tree'*, with nodes and arcs representing, respectively, markings† of the *Petri Net* and the possible changes in state due to transitions firing (see Karp and Miller [*KARP69*], Keller [*KELL72*]). One should notice, however, that the reachability set of a marked *Petri Net* is often *infinite* and consequently a finite representation of it implies a *'many-to-one'* markings mapping onto the same node of the tree.

Since it is out of the scope of this Thesis, the author will not proceed into further details of how this reachability tree is utilized for the *Petri Nets* analysis. We must, however, underline the fact that although many general questions (e.g. *boundedness*, *safeness*, *coverability*‡, etc.) can be answered by this tree, there still exist some other more general questions (e.g. liveness, reachability, etc.), which are not answerable by the reachability tree; in addition, some *Petri Net* problems (e.g. the *subset*, the *equality* problem, etc.) are not solvable (*undecidable*) despite their apparent similarity to the reachability problem (see Baker [*BAKE73*], Hack [*HACK75*]).

---

† *The 'root' of the tree is labelled with the initial marking.*

‡ *The 'coverability' problem is the following: 'Given a marked 'Petri Net' and a marking 'M', does there exist a marking 'M'', in 'r(M)', such that 'M''≥'M'?'.*

From the determination of the actual usefulness of *Petri Nets* aspect, a very significant factor, under intense investigation, is the *computational-complexity* of the reachability problem; in fact, it cannot be exactly estimated, but its lower bounds can be determined.

Lipton has shown that the reachability problem is *exponential time-hard* and *exponential space-hard* (see Lipton [*LIPT75*]); in other terms, the amount of time and memory space required to solve this problem must be, at least, an exponential function of the length of the input description of the *Petri Net* (in the 'worst' case). This is a lower bound; in fact, the actual complexity could be much worse, which makes the cost of answering even simple questions quite significant and such an analysis unfeasible.

In the following of this paragraph, we shall, in particular, emphasize and exemplify the way that most common programming language constructs can be illustrated by utilizing *Petri Nets*.

Although *transitions* and *places* can be, simply, introduced on each arc and node of a flowchart, respectively, to obtain the *Petri Net* program representation, however, their basic interpretation is not quite followed. In fact, in such a modelling, *transitions* will correspond to *executable* statements, while *places* will act as *deciders*, besides having their usual meaning of condition holders. In *Figure (III.A.2.1.2-f1)* a *Petri Net* representation of a *'DO-WHILE'* and an *'IF-THEN-ELSE'* is illustrated.

In addition, *Petri Nets* may represent the *partitioning* of tasks. For example, the *ALGOL*-like program in *(III.A.2.1.2:1)* is illustrated by the *Petri Net* in *Figure (III.A.2.1.2-f2)*.

```
Parbegin
   S1: Parbegin
         S1.1: Begin...End;
         S1.2: Begin...End;
       Parend;                                   (III.A.2.1.2:1)
   S2: Begin...End;
   S3: Begin...End;
Parend
```

On the other hand, *Petri Nets* are extremely valuable for representing properties, such as, process *synchronization* and *mutual exclusion*. For example, a *semaphore* can be represented as an input *place* shared by the *transitions* (*critical sections*), which are to be mutually exclusive. This approach is illustrated in *Figure (III.A.2.1.2-f3)*.

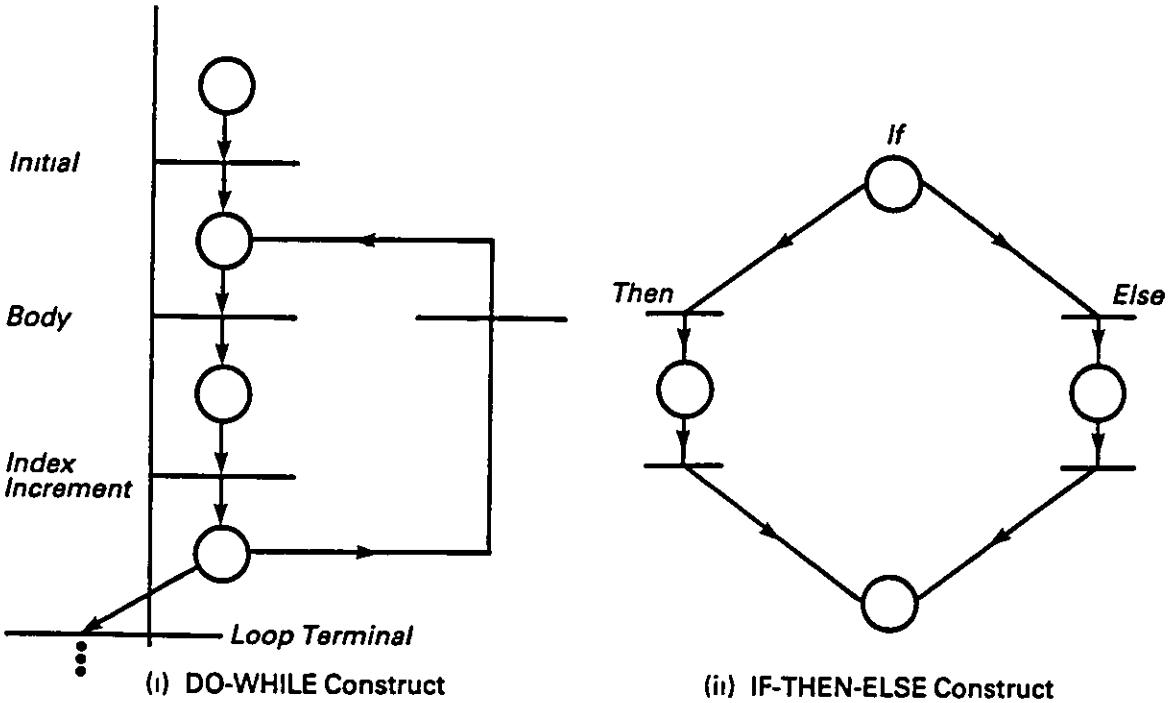


Figure III.A.2.1.2-f1: Programming Language Constructs Representation via *'Petri Nets'*.

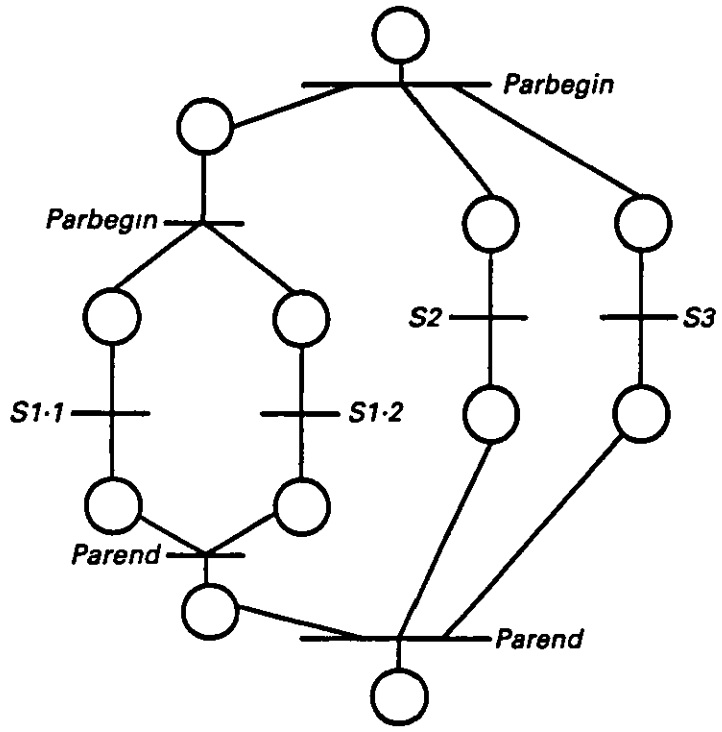


Figure III.A.2.1.2-f2: *'Petri Net'* Modelling of Parallelism.

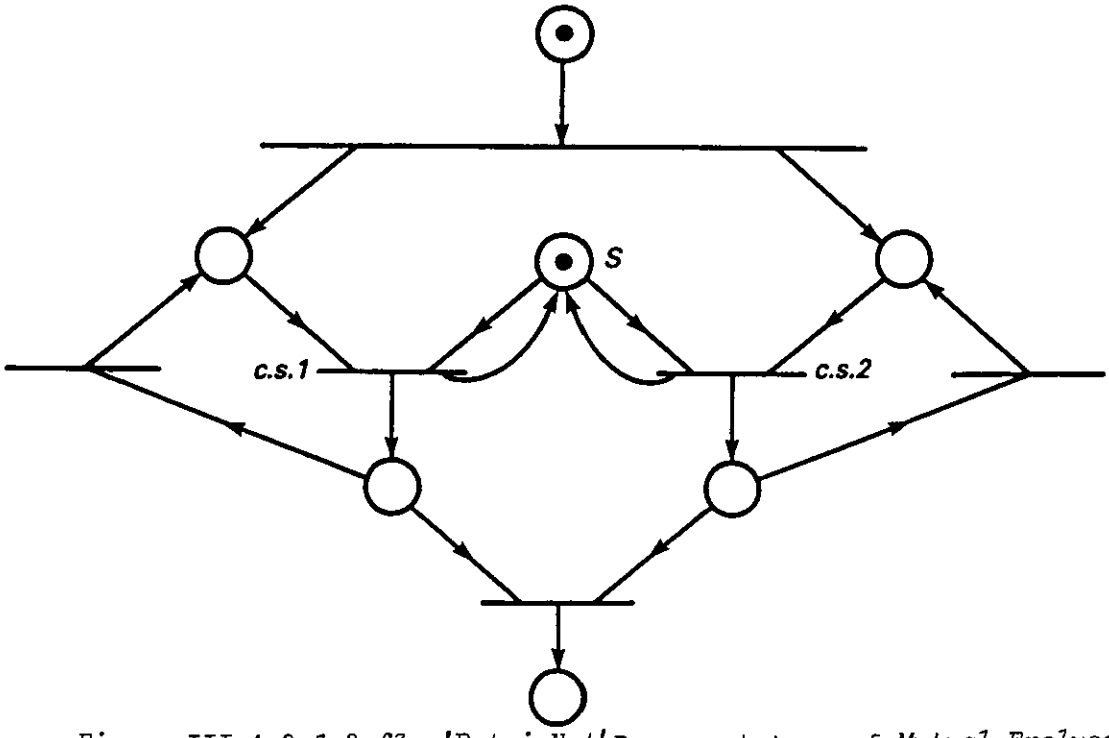


Figure III.A.2.1.2-f3: *'Petri Net'* Representation of *Mutual Exclusion*.

Finally, in a very brief introduction, an important area in which *Petri Nets* have been utilized is the study of *formal languages* (see Peterson [*PETE76*], Hack [*HACK75a*]). To represent the actions of the modelled system, the *transitions* of the *Petri Net*, since theoretically a *finite* number, are labelled utilizing a *finite* alphabet set Σ. In accordance, a labelled marked *Petri Net* defines a set of *strings* over Σ, each string corresponding to a possible execution in the net. The set of all possible strings defines a *Petri Net Language*.

Several varieties of *Petri Net* languages result from slightly different definition approaches. For example, different labelling policies create an entire group of *Petri Net* languages (e.g. λ-*free* languages, etc., see Peterson [*PETE77*], p.243).

Another important determinative factor of *Petri Net* languages is the definition of the set of *final states*; in other terms, since the execution of a labelled *Petri Net* commences from an initial marking (or, one of a *finite* set of initial markings), and terminates in any element of a set of *final markings*, different classes of languages (in fact *four*, namely: *L-type*, *G-type*, *T-type*, and *P-type*), correspond to different definitions of the set of final markings.

To conclude, the original impetus for studying *Petri Net* languages intended to attempt settling some of the decidability questions for *Petri Nets*; however, although there appears a plethora of different approaches to the study of *Petri Nets* by the utilization of formal language theory (see Keller [*KELL72*], Crespi-Reghizzi and Mandrioli [*CRES74*]), much further research is required in this crucial area.

III.A.2.2: EXTENSIONS, SUBCLASSES AND RELATED MODELS TO '*PETRI NETS*'

The acceptance and success of any computational schema, mainly, depends on two principal factors, generally working at cross purposes, its *'modelling'* and *'decision'* power, respectively. Behind the *modelling* power term hides the schema's ability for a faithful representation of the system to be modelled; while, behind the *decision* power term hides the schema's ability to analyze particular model versions determining properties of the modelled system.

The *Petri Net* models, to be specific, appear as a compromising attempt between these two factors; as a matter of fact, a searching answer to the limited *modelling* power of *finite-state* models offered the right motive for the devise of *Petri Nets*, which would attempt not only to improve this *modelling* power, but also, hopefully, to retain most of *finite-state* models *decision* power. However, not all researchers have been thoroughly satisfied with the achieved *modelling power*[†] of *Petri Nets*, therefore, several proposals have been introduced anew in an attempt to increase this power even more.

- *Extensions of Petri Nets*

One simple extension to *Petri Nets* was their immediate extension to *k-safe*; in other terms, more than one *token* would be allowed in a *place*, and thus, a backlog of *tokens* could be built up to be utilized by later firings.

Another primary and significant extension was to remove the constraint that a place could contribute or receive only one *token* from the firing of a *transition*. This was achieved by allowing *multiple*

[†] *Since the correct modelling of relatively reasonable systems was still impossible (see Agerwala and Flynn [AGER73]).*

arcs between *transitions* and *places*, signifying the number of *tokens* required; this class of *Petri Nets* have been called *'generalized' Petri Nets* (see Hack [*HACK74*]), and principally are equivalent to ordinary ones.

A more fundamental extension of *Petri Nets* was due to a major limitation in their *modelling* power, which was their incapacity to *'count'* the number of *tokens* in a *place*, or equivalently, to test whether a place is *empty*. This extension involved the so-called *zero-testing* (see Keller [*KELL72*]); in other terms, it meant the introduction of arcs, from a *place* to a *transition*, which would allow the 'testing' of emptiness of a place, i.e. a *transition* would fire only if the *place* contained zero *tokens*. These special arcs, called *'inhibitors'*, were introduced by Agerwala and Flynn [*AGER73*] and they have been denoted in various ways[†]; this extension has increased the *Petri Nets modelling* power to almost the power of a *Turing Machine*.

Many other extensions of *Petri Nets* were introduced, some of them being:

(i) 'Disjunctive logic' (in contrast to the *conjunctive logic* of *transition* firing), which greatly enhances *Petri Nets* descriptive power. *Disjunctive logic*, instead of a *NOT* condition represented by an *inhibitor*, allows the decision making events to be modelled as *transitions*, itself being denoted by a '+' at the input or output of the *transition*.

(ii) 'Token absorbers', useful for 'killing' redundant processes and for leaving the net in a 'Properly Terminating' - *PT* condition (see

---

[†] *Either as arcs with a 'dash', or as arcs with a circle at the oriented end.*

Baer [*BAER82*]).

(iii) 'Coloured tokens', to model program reentrancy and the instantiation of several identical processes (see Jensen [*JENS79*], Baer and Jensen [*BAER77*]). In this extension *places* contain bags of *coloured tokens* and are connected to *transitions* via labelled arcs, the labels being either sets of colours, or free variables. In accordance, the firing rules are modified as follows: Assuming a (p_i,t)[†] being labelled with a set N_i, then t can be enabled only if p_i contains, at least, one *token* from each colour belonging to N_i and the firing will remove one *token* from each colour; if (t,p_j) is labelled with N_j, then the firing of t will deposit on p_j one *token* from each colour belonging to N_j.

In conclusion, rather than carrying on discussing other extensions, such as, time-bounds on *transition* firings, or other constraints on the subsets of *places*, which can be full concurrently or timing attributes for *transition* firings, the author will emphasize on the fact that, in terms of *modelling* power, *Petri Nets* seem to be just below *Turing Machines*, so that any significant extension would result in *Turing Machine* equivalence (see Peterson and Bredt [*PETE74*]).

- *Subclasses of Petri Nets*

The motive behind the definition of these subclasses was a compensational increase in *decision* power to balance the *modelling* power limitations. More specifically, for *Petri Nets* many decision problems were equivalent to the *reachability problem*, whose complexity has shown to be (albeit its *decidability*) very difficult to solve; so *Petri Nets* might be too powerful to be analyzed, a fact which consequently leads

[†] *The notation is:* p_i*-'*i^{th}*-place'*, t*-'transition'*.

to the definition (with restrictions on their structure for better analyzability) of a number of *Petri Nets* subclasses, in the hope of finding a subclass of known *decision* power for practical purposes.

In particular, two subclasses are most commonly considered, the *'State Machines'* and the *'Marked Graphs'* (see Holt and Commoner [*HOLT70*]).

The *State Machines* are restricted *Petri Nets*, so that each *transition* has exactly one input and output, being obviously *conservative* and hence *finite-state*; in fact, they are exactly the class of *finite-state* machines. This finite property of them results in a significantly high *decision* power, but they are of limited usefulness in modelling infinite systems.

On the other hand, a *Marked Graph* is a *Petri Net* in which each *place* has exactly one input and output *transition*. Various algorithms exist which can prove that a *Marked Graph* is *live* and *safe* and the *reachability problem* is solvable on them. Therefore, *Marked Graphs* have high *decision* power, but a limited *modelling* one, since they are restricted to model systems without control flow branches; in other terms, parallel activities can be easily modelled, while alternative ones cannot.

In addition to these two classes, there are some others like, the class of *'Simple' Petri Nets*, the class of *'Conflict-free' Petri Nets*, the class of *'Persistent' Petri Nets* (see Landweber and Robertson [*LAND75*]) and the class of *'Free-choice'* ones (see Hack [*HACK72*]). In particular, Hack has shown that the class of *Free-choice Petri Nets* can quite successfully model a class of systems, called *production schemata*, which are similar to assembly-line systems.

- *Related models to Petri Nets*

In this part we must emphasize on *'Vector Addition Systems'*, which were defined by Karp and Miller [*KARP69*] and are equivalent to *Petri Nets*.

A *Vector Addition System* is essentially a mathematical formulation, in terms of vectors, of the *markings* and *transitions* of a *Petri Net*, in an attempt of a more formal manipulation of the nets. A generalization of these systems produced the related and equivalent model of the *'Vector Replacement Systems'* (see Keller [*KELL72*]).

The author wishes to point out the fact that *Petri Nets* certainly are far from being the only model of concurrent systems to have been developed. There are many other models developed to-date, including, *'Program Graphs'* (see Rodriguez [*RODR67*]), *'Computation Graphs'* (see Karp and Miller [*KARP66*]), *'Message Transmission Systems'* (see Riddle [*RIDD72*]), *'Flow Graph Schemata'* (see Slutz [*SLUT68*]), *'Complex Bilogic Directed Graphs'* (see Gostelow [*GOST71*]), etc.†

To conclude, the reader should be reminded that, this particular introduction to *Petri Nets* is justified from the fact that these nets consist of the 'Kernel' of all the above models. Comparisons of the properties of many of these models (see Peterson and Bredt [*PETE74*]), have shown that most of them are either subclasses of *Petri Nets*, or are equivalent to them, of course under a certain predefined notion of the equivalence‡ property; consequently, this exclusive reference to *Petri Nets* has the form of a tribute to the fundamental offer of this pioneering schema.

---

† *Baer [BAER73] has published a survey of some of these models.*

‡ *Lipton, Snyder and Zalcstein [LIPT74] found important differences in the 'modelling' power amongst the various models of concurrent systems, by utilizing a considerably different (but equally valid) definition of equivalence.*

III.A.3: Objectives Of 'Fifth Generation Computer Systems' - *FGCS* And Novel Decentralized Machines As Their Potential Architectural Basis

Although researchers, from all over the world, have already started competing in what could be the most significant scientific race of this century - the race to develop the 'Fifth Generation Computer System' - *FGCS*, the path to that goal is far from being straight-forward; the difficulties are almost as daunting as the prospects are exciting.

The term *Fifth Generation*, grossly, understates the task volume, by implying a continuation in progress from the first *four* generations, which have driven computers from *valves*, through *transistors* and *microchips*, to extremely powerful microchips. This new generation will be proved to be more than just a great leap ahead, since it is a leap in the dark.

In other words, the *Fifth Generation* in the computing domain, for parallelization, it is analogous to attempting a leap of similar size with the one made in aeronautical science with the commencement of flights into space. The progress and improvement there, from the Wright brother's first plane, to a modern jumbo jet, has been made steadily depending on the same air travel principles; flights into space, on the other hand, introduced a much more different and advanced technology.

The forthcoming *computers generation* will mean much the same thing for computing: A complete re-think and a lot of innovative technology.

The principal difference, between the up-to-date known computer systems and those of the *Fifth Generation*, lies in that the former perform exactly what they are asked to, following a pre-arranged track; the aim of the next generation will be, as much as possible, the

acquisition of knowledge and its *intelligent* utilization. In other terms, *FGCS* will be built to operate more like a human brain and the operators will be able to interrogate them quickly discovering their line of reasoning.

Although the realization of the *Fifth Generation* concept requires a whole series of technological and scientific advances, already there are a few *'Proto-Fifth'* computer systems designed to explore certain aspects[†] of this generation. In particular, British pioneering has exhibited some strengths in *Fifth Generation* hardware research, by having developed by *INMOS* the so-called *transputer* (see also *par.-II.A.2*), which is the first computer on a single chip, including memory as well as a microprocessor and a number of communication links, which will allow direct connection to other transputers (see May and Shepherd [*MAYS84*]).

The *transputer* is planned to be utilized in the *Alice* experimental computer, which is currently being developed at Imperial College and is primarily designed for *inferencing* utilizing parallel operations. This *Alice* computer in turn, will be, hopefully, utilized in the *Plessey* project, which aims to produce a speech recognition system, with a *5000* words vocabulary, a minimal error rate, and the ability to adapt to different speakers.

In general terms, for the *FGCS* to succeed in achieving the above set goals, *five* key-problems have to be cracked, concerning, the 'Very Large Scale Integration' - *VLSI*, the *Logical Inference*, the *Storage*, the *Parallel Processing* and finally the *Natural Language* to be utilized.

[†] *However, the real problem will be to combine all these aspects into one all-embracing super-machine.*

The silicon-based *VLSI* construction of devices will offer the potential of placing vast amounts of computer power into a tiny area, since the forthcoming generation will require chips carrying quite a few millions of computing elements. In accordance with the present pace of progress, one could easily attempt the prediction that, at the end of this decade it would be feasible to place the equivalent of today's huge computers on a few tiny chips; any such prediction, however, probably will not come true due to the fact that design and fabrication problems grow disproportionately with increasing chip complexity.

The *Logical Inference* problem underlines the computer's ability to *think* for itself; however, this will require novel architectures, software, and special languages for the programming, which will need to be not *'procedural'* like *Fortran*, *Cobol*, etc., but *'declarative'* (i.e., consisting of statements) like *Prolog*, *Lisp*, etc. The advantage of such languages is that the programmer, instead of instructing the computer the solution steps, simply feeds it with all the required knowledge for the particular subject (i.e. the known facts, and the rules and relationships connecting them), informing it what is to be performed, not how to perform it.

However, from the *Storage* point of view, this new generation of computers will have to manage huge amounts of information in a useful form, which will require a similar to the human brain flexible approach. Today's Databases can cope with a few thousand items and the associations amongst them; the *FGCS* will be aiming at *Knowledge Bases* comparable in size to the Encyclopaedia Britannica.

The solution of the difficult *Parallel Processing* problem should,

eventually, not only produce huge increases in processing speed, but also to minimize the great difficulties of keeping track of many operations going on simultaneously; and this is a vital key-problem to crack, since there lies the crucial difference which will make the forthcoming generation to perform much more like a human brain.

Finally, the last problem, which is required to be tackled, concerns the ability of the *FGCS* to understand things in terms similar to those we use in real life; in other words, it implies the requirement of a *Natural Language*. This general understanding of what one really means when 'speaking' to a computer, forms one of the most important problems facing researchers around the world; since, it involves such problems as, how word order affects meaning, ambiguities resolution, metaphors and intentions recognition, etc.

In a more pragmatic and determinate manner, reviewing the principal aspects of *FGCS*, at first, one can easily realize that each system will consist of a network of computing elements[†], to provide either a general-purpose or a special-purpose function, ranging in power from a main-frame computer, to a miniature microcomputer; while, on the other hand, that the main objective of *FGCS* is the realization of the so-called 'Knowledge Information Processing Systems' - *KIPS*, which primarily implies the development of a new software for *inference* computing capable of handling vast amounts of data.

As a first partial conclusion, in relation to the former aspect, for *FGCS* to be programmed as individual computers and for their computing

---

[†] *Either physically close, as on a single highly integrated chip, or dispersed across a building or country (see also par.-I.A.1).*

elements to cooperate together, it is necessary to exist a single computer architecture to which they will all conform.

More analytically, the major components of *KIPS* are, the *Inference Engine* and the *Knowledge Base System.*

Although *inference* operations can be implemented on a conventional von Neumann computer, an effective model for parallel execution is required, to perform *inference* operations of practical size with high-speed. A variety of different computational models[†] have been proposed as the most eligible for being the architectural basis for *FGCS* - namely, *Data Flow*, *Control Flow*, *String Reduction*, *Graph Reduction* models, or even a 'synthesis' of them called *Recursive Control Flow* model.

More specifically, in 'data-driven' (e.g. *Data Flow* and 'multi-thread' *Control Flow*) models, the availability of operands triggers the execution of the operations to be performed on them, while in 'demand-driven' (e.g. *Reduction*) models, the requirement for an operand triggers the operation(s) that will generate it. Since a comprehensive covering of all these models and their underlying concepts will be too extensive, the author will limit his reference to the *Data Flow* model only, which seems to be one of the most attractive and feasible solutions for the above problem; however, the particularly interested reader can refer to Treleaven and Hopkins [*TREL81*] and Treleaven [*TREL82*] for a specific analysis of all these models and concepts.

On the other hand, *KIPS* require knowledge information of a large size; consequently, to realize a *Knowledge Base System*, with such a high performance[‡], sophisticated hardware support is required, a powerful

---

[†] *These models are distinguished by the way computations manipulate their arguments and by the way the execution of computations is initiated.*

[‡] *Since it is required to be highly general, so as to handle any kind of knowledge data.*

means of which can be proved to be the so-called *Data Base* machine. More specifically, such a machine will be utilized for storing and accessing information, while the *Inference Engine* for drawing conclusions. However, an additional difficulty will be due to the requirement for the development of special software for managing the *Knowledge Base*, which will not just be a usual Database containing only the known facts, it will include the rules and relationships connecting them, as well.

Accordingly, we can conclude that the *Data Flow* and *Data Base* machines, which we briefly introduce in the following paragraphs, are the most promising candidates for the basic architecture of *KIPS*. They will, eventually and hopefully around the end of this decade, have been combined, via a high-speed local network†, in a working prototype, namely a functionally distributed architecture‡, which will probably bear the name of a *'Superinference'* machine (see Sakamura, et al [*SAKA82*]); however, since the imminent general plan is the establishment and experimentation of the new technology in limited areas, rather than producing an intelligent system with a full Knowledge Base, it is the author's opinion that research should concurrently start on network architectures, which may offer the basis for further systems growth.

---

†*The transfer rate of this network will probably be 100Mb/s at the initial stage, to be improved, later on, to 1Gb/s.*

‡*Remote accesses to this system will be possible through a suitable 'Fifth Generation' communication network.*

III.A.3.1: ORIGINATION, FUNDAMENTAL HARDWARE AND SOFTWARE PRINCIPLES, CHARACTERISTICS OF THE '*DATA FLOW*' MACHINE ARCHITECTURES

As *inference* is the fundamental operation in *KIPS* and 'trial and error' and non-deterministic operations will be the characteristic of Inference Engines, and since the *Data Flow* model was suggested as a suitable parallel hardware to form their basis, a *FGCS* project should commence with an evaluation and feasibility study of the existing *Data Flow* machine architectures.

Although one cannot precisely define originators of 'data-driven' computation, it appears that its theoretical basis was set in the paper of Karp and Miller [*KARP66*], in 1966. A couple of years later (1968), J.B. Dennis and, almost at the same time, Tesler and Enea commenced their data flow research; the former, defined graphs allowing the expression of algorithms by explaining data dependency only, while the latter, in a report they published (see [*TESL68*]), were concerned with programming languages embodying syntactic and semantic features for data flow programming. The particularly interested reader, for a comprehensive historic evolutional chronicle, should refer to Syre [*SYRE82*], who presents, although not complete, a relative amount of data flow studies in the form of a diagram covering the period 1968-80.

To cover the way machine code programs are represented and executed in a *Data Flow* machine architecture, the author will utilize the term *program organization*; in fact, there are two fundamental computational mechanisms concerned with the program organization, the *data* and the *control* mechanisms.

The *data* mechanism defines the way a particular argument is utilized

by a number of instructions, while the *control* mechanism defines how one instruction causes the execution of one or more other instructions, and also the resulting control pattern.

A data flow program organization has a 'by value' *data* mechanism and a 'parallel' *control* mechanism. More explanatorily, a 'by value' *data* mechanism means that an argument, generated at run-time, is shared by replicating it and giving a separate copy to each accessing instruction, this copy being stored as a value in the instruction; a 'parallel' (or, 'by availability') *control* mechanism means that the control signals the availability of arguments and an instruction is executed when all its arguments (e.g. input data) are available. For comparisons with the other computational mechanisms underlying the models mentioned in (*par.-III.A.3*), the author refers the interested reader to Treleaven [*TREL82*].

In a data flow computing environment, information items appear as *operation packets* and *data tokens*. In particular, data flow programs are represented by *directed graphs*[†], held in the memory section, which show the flow of data between *instructions*; the *arcs* of these data flow graphs are queues of *data tokens* directed from one *operator node* to another. Each instruction consists of an *operator*, one or two *operands*, and one (or more) *destinations* to which the partial result (i.e. *data token*) will be sent, all these forming the so-called *operation packet*. Many of these packets or tokens are passed amongst various resource sections in a *Data Flow* machine; therefore, such a machine can assume a *packet communication* architecture, which is a type of distributed

[†] *For their definition refer to (par.-III.A.2).*

Multiprocessor organization. In *Figures (III.A.3.1-f1,f2)*, respectively, are illustrated the instruction execution mechanism, and *three* data flow graph snapshots of a computation, where data tokens are represented by *black dots*. *Data Flow* machine architectures, depending on the way of handling data tokens, are distinguished into the *'Static'* and the *'Dynamic'* models, the delineation of which is introduced in *Figures (III.A.3.1-f3,f4)*.

More specifically, in a *'Static' Data Flow* machine tokens are not labelled and *control tokens* must be utilized to acknowledge the proper timing in transferring data tokens between 'nodes'; consequently, since successive sets of tokens could not be distinguished, only one token is allowed to exist on any arc, at any given time. J.B. Dennis and his research team at the MIT Laboratory for Computer Science are currently developing such a computer model.

On the other hand, a *'Dynamic' Data Flow* machine utilizes *tagged tokens* (labelled or coloured), to allow multiple tokens to appear simultaneously on any input arc of an operator node. Through this tagging, the context of each token can be uniquely identified and, although

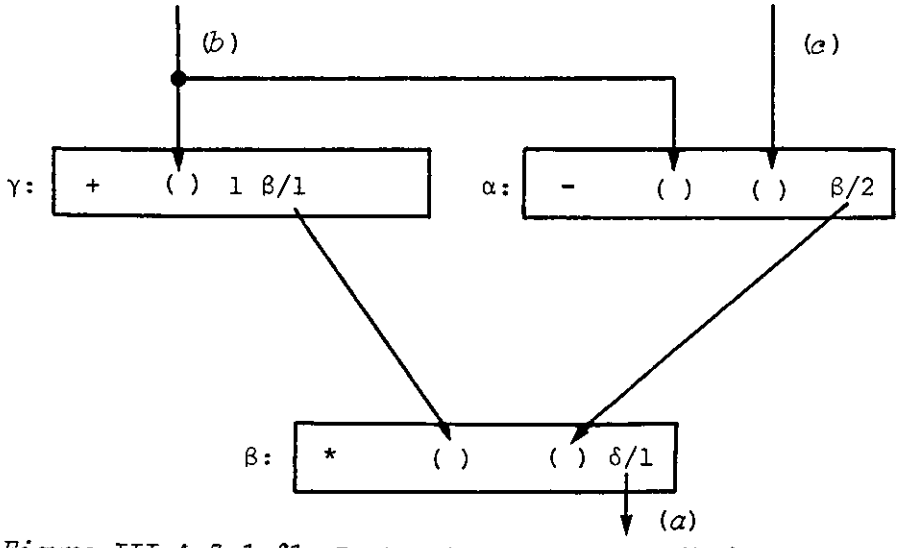


Figure III.A.3.1-f1: Instruction Execution Mechanism in a *Data Flow* Machine for the Computation of $a=(b+1)*(b-c)$.

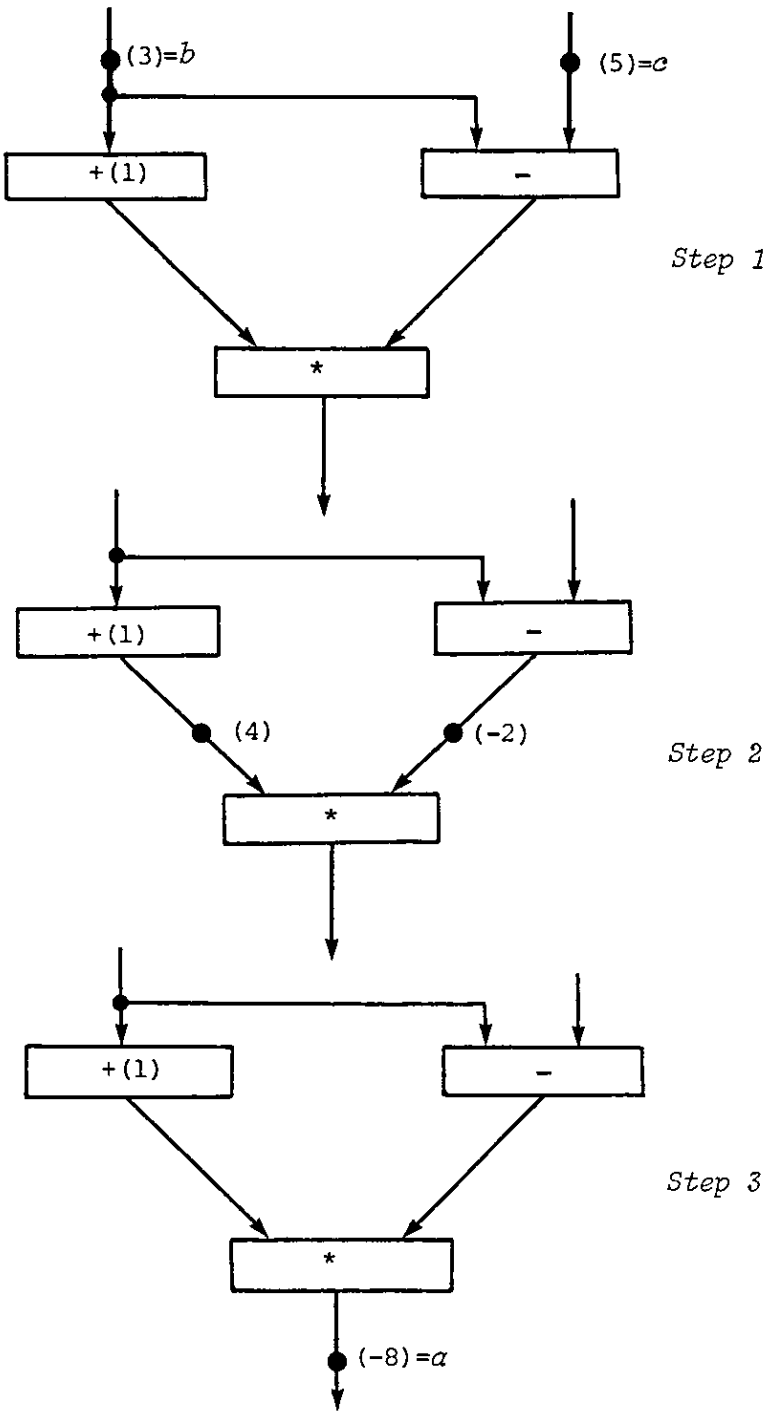


Figure III.A.3.1-f2: Three Snapshots of the Data Flow Computation for $a=(b+1)*(b-c)$.

additional hardware is required to attach these tags and to perform tag matching, no control tokens are required to acknowledge the transfer of data tokens amongst instructions. Such a dynamically tagged data flow

model suggests that maximum parallelism can be exploited from a program graph.

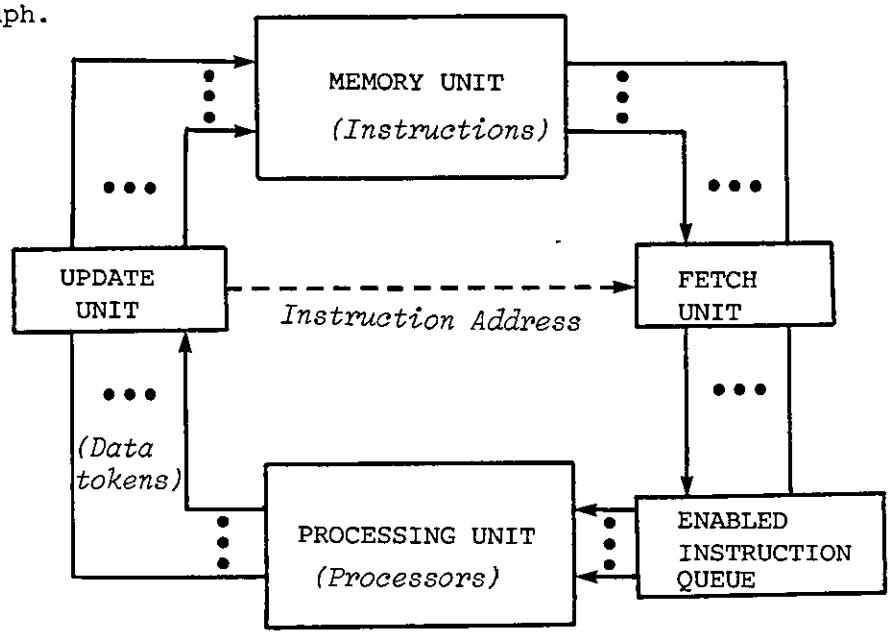


Figure III.A.3.1-f3: A *'Static' Data Flow* Machine Organization.

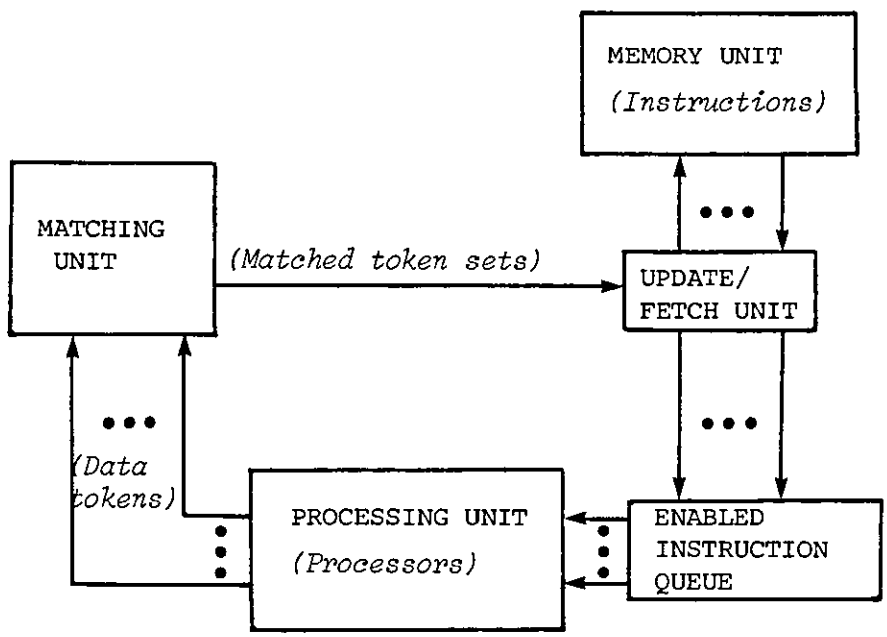


Figure III.A.3.1-f4: A *'Dynamic' Data Flow* Machine Organization.

If the graph is *cyclic*, the tagging allows dynamically unfolding of the iterative computations. *Dynamic Data Flow* computers include, the Manchester machine developed by Watson and Gurd, at the University of Manchester, and the Arvind machine, under development at MIT, which has evolved from an earlier data flow project at the University of California, at Irvine.

Although both these packet communication organizations comprise multiple processing elements, capable of an independent and asynchronous executable instruction packets evaluation, they are based on two different schemes for instruction execution synchronization, the so-called *'Token Storage'* and *'Token Matching'* schemes.

In the former scheme, single tokens from the input pool of the *Update Unit*, through this unit, are actually stored into their destination instruction, or a copy of it, in the *Memory Unit*; an instruction is *enabled* to be forwarded, through the *Fetch Unit*, to the *Enabled Instruction Queue*, when it has received all its required operand tokens.

In the latter scheme, the *Matching Unit* is utilized to group together and temporarily store tokens taken from its input pool, destined for the same instruction. When the group is complete it is released to the *Fetch/Update Unit*, which in turn forms the *enabled* instructions, by merging the values from the token sets with a copy of their instruction, and forwards them to the *Enabled Instruction Queue*.

In conclusion, both, *Static* and *Dynamic Data Flow* machine architectures, have a *pipelined ring* structure. In general, this ring comprises *four* resource sections: *Memories*, *Processors*, *Routing Network*, *I/O Unit* , but it can be certainly extended to many other improved *Data*

Flow architectural configurations.

In order to identify and briefly clarify the role of these structural components on some actually implemented *Data Flow* machine architectures, the primary aim of the following paragraph will be the introduction of two characteristic representatives of the above categories: The Dennis *Static* machine at MIT and the Manchester *Dynamic* machine in England.

III.A.3.1.1: PROTOTYPE '*DATA FLOW*' MACHINE ARCHITECTURES, PROGRAMMING LANGUAGES, AND FURTHER DESIGN ALTERNATIVES

The intent under the *Data Flow* machine architectures presentation spectrum is the laying out, in a comparative-like manner, of the architectural concepts of the Dennis *Static* machine and the Manchester *Dynamic* machine, rather than any description of their implementation details; the outline of both architectures is introduced in *Figures (III.A.3.1.1-f1,f2)*.

The Dennis machine is designed to exploit the concurrency in programs represented by *Static* data flow graphs. In this machine architecture one can identify *five* major sections, which, being connected by channels through which information flows, can operate independently without utilizing central timing signals:

- *Memory Section*, which is subdivided into 'cell blocks' (each with a unique address, the cell *identifier*) to hold instructions.
- *Processing Section*, which consists of processing units that perform functional operations on the data tokens.

- *Arbitration Network*, which transmits operation packets from the memory section to an appropriate processing unit according to the operation code of the packet.
- *Control Network*, which delivers a control token from the processing section to the memory section. These tokens act as 'acknowledge signals', when data tokens are removed from output arcs, to correctly implement the firing rule for program graphs; and,
- *Distribution Network*, which accepts data tokens from the processing section and, utilizing the address of each[†], directs them to the correct register of an instruction cell. In fact, each instruction cell is composed of three *registers*; the first register, holds the operation to be performed and the address(es) of the register(s) to which the result of the operation is to be directed, whilst the other two registers, hold the operands to be utilized in the instruction execution.

The Manchester *Data Flow* machine, also, demonstrates *five* functional blocks communicating in a clockwise direction around a ring:

- *Switch*, which handles external input/output.
- *Token Queue*, which is a 'First-In, First-Out' (*FIFO*) buffer to equalize data rates around the system. The token package is the main unit of information, as in the *Static* model, comprising a label in addition to the data value and destination pointer.
- *Matching Store*, which although it is implemented using a conventional random access store with hardware hashing techniques, it is

[†]*Each result token comprises a result value and a destination address derived from the instruction being processed by a processing unit.*

'associative' in nature. The associative field is formed from a concatenation of the label and next instruction fields, the value field being the token value. In the case of single input nodes, where no matching operation is required, a control digit in the next instruction information allows a bypass of this unit.

- *Instruction Store*, which is a random access memory holding the directed graph description. Each entry is in the form of a nodal operation and the addresses† of the subsequent nodes to which the data token(s) will be directed; and,
- *Processing Units*, which are microprogrammed microprocessors with a 'distribution' and 'arbitration' system. The former system, on receipt of an executable package selects any processor which is free and allocates the nodal operation, whilst the latter system controls the output of tokens from the processors.

The delineate presentation of the *Data Flow* machine architectures would be incomplete without a brief outline of the methods available for programming them, albeit no attempt will be made to address the problems of doing this; after all, the *Data Flow* machines are language-oriented machines. In fact, their research started with data flow languages and it is the rapid progress in *VLSI* that has pushed the construction of several hardware *Data Flow* prototypes in recent years.

It would seem that only *graphical* languages (see Dennis [*DENN74*]) would offer a natural way of expressing a directed graph; however, *textual* languages are far more familiar and it can be argued that

† *In fact, in this implementation, nodes are able to specify two output destinations for their result.*

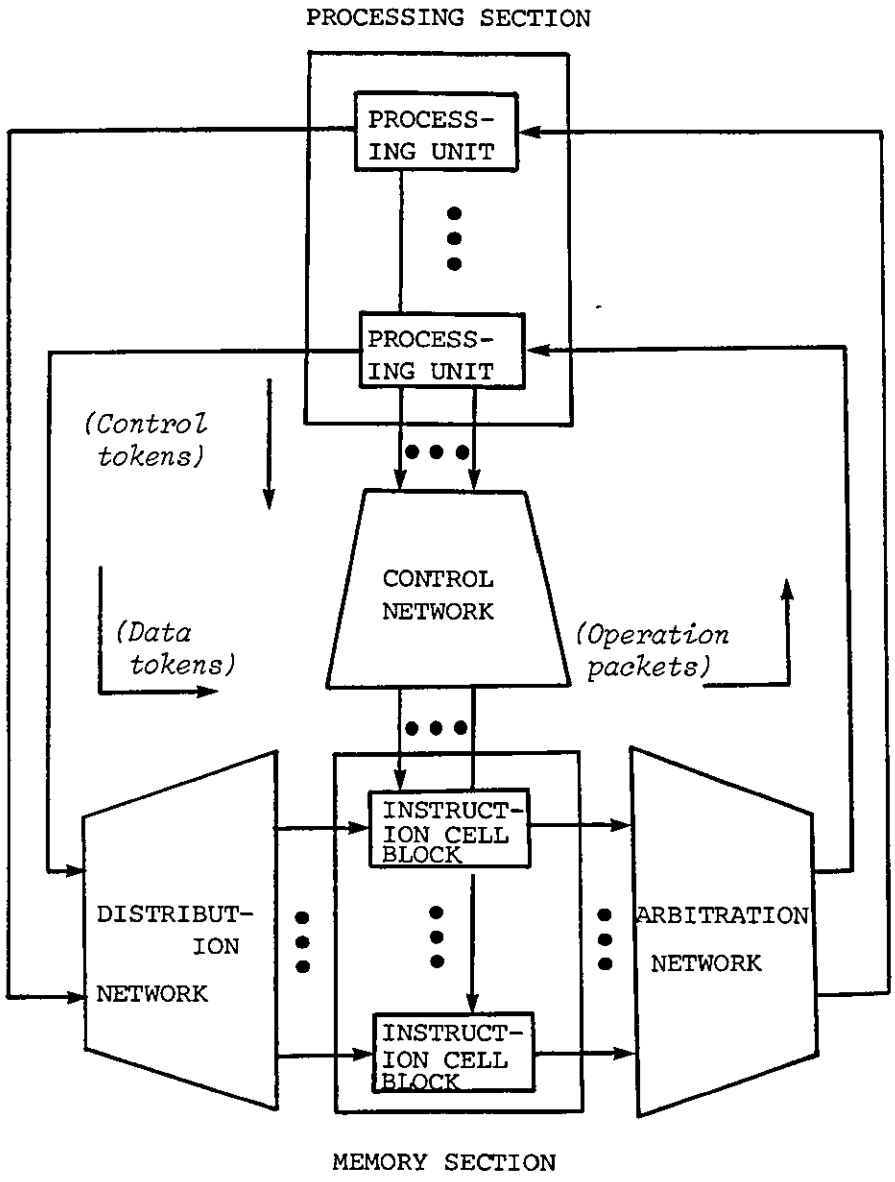


Figure III.A.3.1.1-f1: The Dennis *'Static' Data Flow* Machine Architecture at MIT.

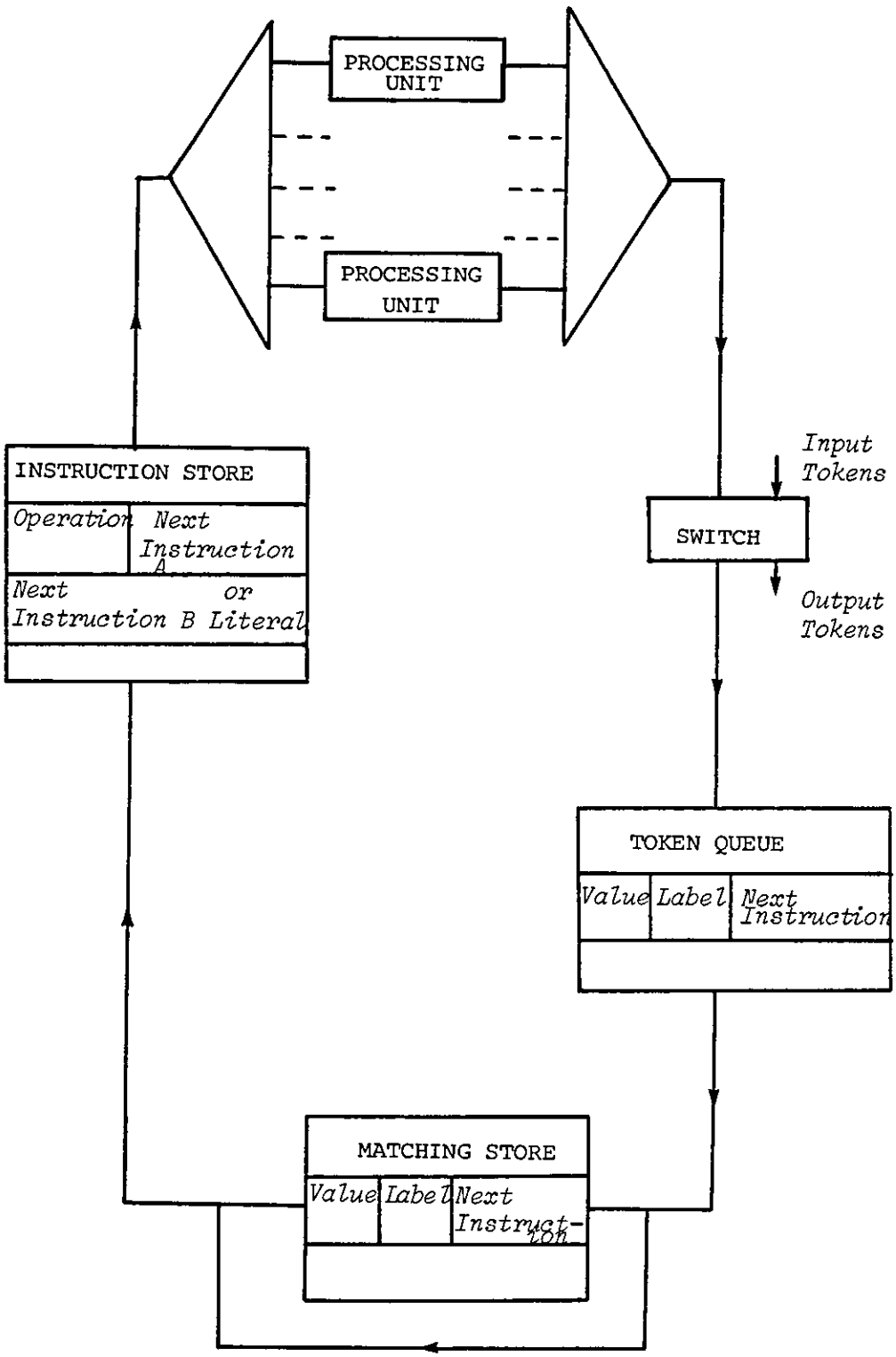


Figure III.A.3.1.1-f2: The Manchester *'Dynamic' Data Flow* Machine Architecture.

features such as data structures are easier to express into a textual form.

One approach would be to consider a conventional language and translate it into a data flow graph. The principles involved in such a translation were originally suggested by Miller and Rutledge [*MILL66*].

Another class of languages, the *Single Assignment* languages, are more naturally suited to the expression of parallelism in data flow form.

An efficient data flow language should be able, to express parallelism in a program more naturally, to promote programming productivity, and to facilitate close interactions amongst algorithm constructs and hardware structures. Examples of data flow languages include, the 'Irvine Data flow' - *ID* language and the 'Value Algorithmic Language' - *VAL*, amongst several *Single Assignment* and *Functional Programming* languages that have been proposed by many computer researchers. Other similar languages, such as *LUCID* (see Ashcroft and Wadge [*ASHC77*]), have been developed with emphasis on the proof of program correctness. It is, certainly, too early to forecast which of these approaches will eventually be the most fruitful, since the exact form of languages which will gain acceptance is yet to be decided.

Finally, we must underline the fact that, there are several data flow projects with special architectural approaches, different from the *Static* or *Dynamic* machines mentioned previously. For example, the 'Data-Driven Machine' - *DDM* tree structured architecture, currently located at the University of Utah, the *LAU Data Flow* system in Toulouse, France, with 32 bit-slice microprocessors interconnected by multiple buses, etc.

In fact, up to 1983, only the *DDM* system at Utah, the *Dynamic*

'Experimental system for Data-Driven processor arraY' - *EDDY*[†] in Japan, the Manchester machine and the French *LAU* system, were operational *Data Flow* computers.

Since most of the data flow projects emphasize on the run-time simultaneity at the *instruction level*, which sometimes (because of high system overhead, in detecting the parallelism and in scheduling the available resources) results in a very poor performance, we shall conclude this paragraph with *two* other design alternatives to the data flow approach; they should offer higher machine compatibility, as well as, a better utilization of the existing software assets, the 'dependence-driven' and the multilevel 'event-driven' approaches.

The former approach was proposed by Gajski, et al [*GAJS81*], in 1981. The idea was to raise the level of parallelism to *compound-function* level at run-time, and apply the data flow principles over multiple *compound-function* nodes. A *compound-function* is a collection of computational tasks (e.g. array (vector, matrix) operations, blocks of assignment statements, etc.), that are suitable for parallel processing by multiprocessors. A program is a dependence graph connecting the compound-function nodes; in a sense, this approach is a 'procedure-driven' approach, where traditional high level languages can be utilized instead of data flow languages.

The 'event-driven' approach appeared as a generalization of the former one, made by Hwang and Su [*HWAN83*]. An *event* is a logical activity, which can be defined at the job level, through, down to the instruction level, after a proper *abstraction* or *engrossment*. However,

[†] *The functional language to be utilized in EDDY is called 'VALID' (see Hwang and Briggs [HWAN84]).*

a program abstraction mechanism is required to be developed, which should not require high system overhead. In addition, a hierarchical scheduling of resources is required and this has proved to be the most challenging part of the research in this approach. Heuristic algorithms are required for scheduling multiple events to the available resources; in fact, for the scheduling of the events, this approach considers the utilization of *priority queues* on all enabled activities, instead of the 'First-In, First-Out' (*FIFO*) scheduling policy of a 'data-driven' system.

As a final remark, for research-oriented readers, a number of important issues, that demand further efforts towards the development of workable *Data Flow* Multiprocessor systems, can be found in Hwang and Briggs [*HWAN84*].

III.A.3.2: A General Specification Of Research Subjects And Characteristics For A *FGCS 'DATA BASE'* Machine Architecture

In general terms, a *Data Base* machine can be thought of as either, a 'dual' unit stand-alone system, where one unit is a 'Host' function handling unit and the other a 'Database' function handling unit; or, as just a 'Host' attached functional unit, conceptually, equivalent to the latter unit above. A proposed[†] general configuration for a *Data Base* machine is illustrated in *Figure (III.A.3.2.-f1)*.

From the type of *functions* aspect, that should be assigned to a *Data Base* machine, one should distinguish functions such as, data

[†]*See Tanaka, et al [TANA82].*

manipulation, data compression and encoding, memory management, query analysis and optimization, integrity control, concurrency control, recovery control, etc.

This paragraph, primarily, aims towards a general descriptive presentation of the *Data Base* machine fundamental concepts and characteristics, rather than the analysis of proposed Database organization schemas. However, an effective solution to this problem would be a *Data Base* machine that supports a *relational model* (see Ullman [*ULLM82*]), since through that model any kinds of data can be expressed; as primitive operations could be considered the 'set' oriented operations, such as the ones defined in *relational algebra* (e.g. union, intersection, Cartesian product, etc.), which will form the host interface.

The configuration of the hardware system is a means to realize the parallelism of processing, which can be introduced under three forms.

The *first* form of parallelism refers to the utilization of many processing elements and memory modules, connected to each other through connecting circuits, such as packet switches; it is closely related to the concurrent multi-user support, to permit the simultaneous execution of many user queries, and/or updates, which is essential to improve the system throughput sufficiently enough, to be utilized as the Knowledge Base System for the Inference Engine.

The *second* form refers to the internal parallelism of a processing element, that can be made of many functional units (e.g. *sorter*); it is the main factor to limit the speed of a single primitive operation and is realized by the parallel and pipelined processing amongst many units.

Finally, the *third* form refers to the internal parallelism of a

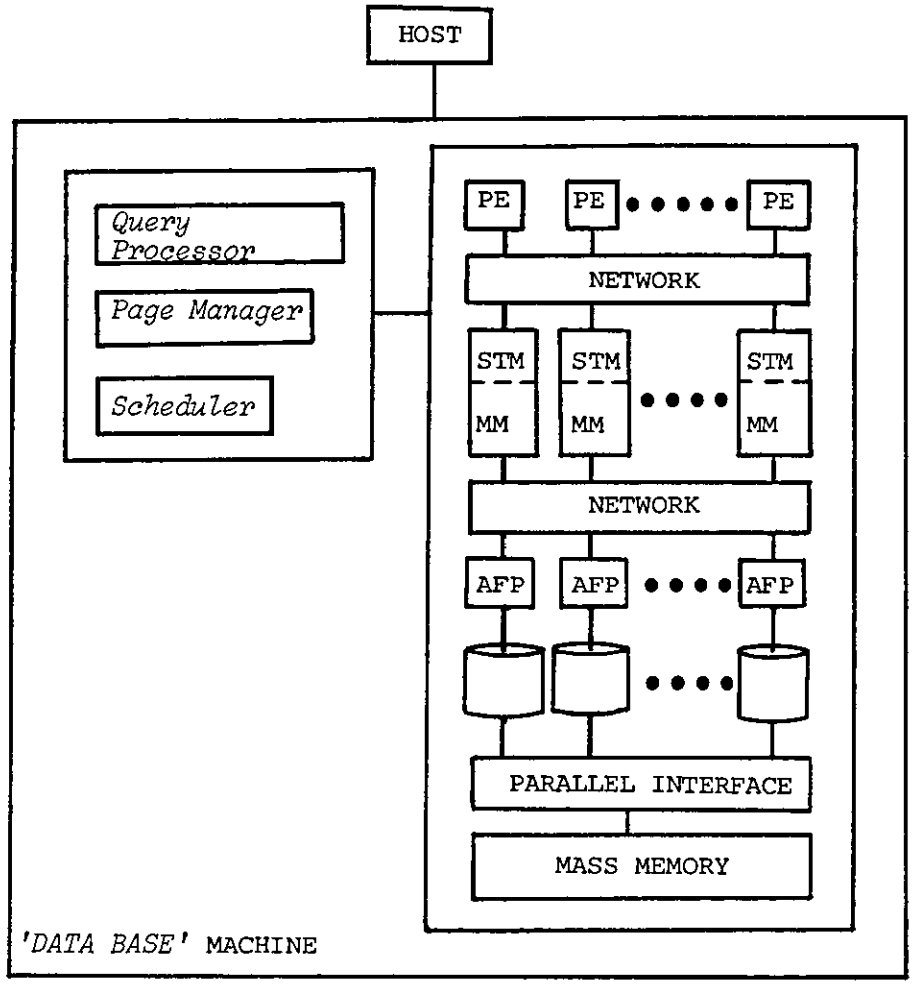
memory module (i.e. 'working memory'), made of many memory cells and several functional units (e.g. *search* modules); it aims substantially to reduce the data volume that flows within the network of processing elements.

To-date, there have been proposed many *Data Base* machine architectures, but almost all of them support a limited data capacity. We should investigate a mechanism of large capacity handling feature, through, probably, *pagination*, or *staging networks* that will load necessary relations to the working memory modules from a mass memory.

'Security' and 'Integrity' support functions are other very important research subjects. The future *Data Base* systems will not be stand-alone systems, but, hopefully, connected to each other by communication lines, thus forming a total 'virtually' integrated data system. However, to achieve this objective, problems such as, data model homogenization, query processing, commitment control against failures, concurrency control of update operations, etc., have to be tackled.

Finally, as the *Data Base* machine architecture will be implemented by *VLSI* devices, the hardware structure had better be modular.

To conclude, another important research subject, and undoubtedly not the last one, concerns the *Reliability* problem, which as the system size will grow, will become severer because of its great influence to the users in the case of a system 'failure'; however, it may require a great overhead to collect the check point information and update records, due to the large data capacity.



PE : *Processing Element*

STM : *Search & format Transformation Mechanism*

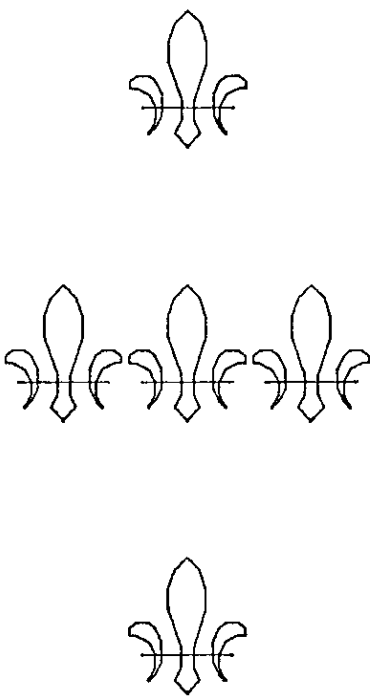
MM : *Memory Module*

AFP : *Associative File Processor*

Figure III.A.3.2-f1: A General Configuration for a *'Data Base'* Machine.

SECTION B

EMBEDDING INFORMATION FLOW SCHEMES ON GRIDS AND IN CHIP AREA AND TIME



'By the Nine Gods he swore it,
And named a trysting day,
And bade his messengers ride forth
East and west and south and north,
To summon his array.'

Lays Of Ancient Rome

Horatius ,1,1842

Lord Macaulay

III.B.1: The Impact Of The Technological Innovation On Future Architectures - The *VLSI* Challenge

The subject of this paragraph, rather than technological innovation in general, will be what is widely believed to be the most important opportunity since the industrial revolution, rivalling it in significance. In fact, futurists foresee that two new and far-reaching technologies, 'Very Large Scale Integrated' - *VLSI* circuits and *Biotechnology*, will dominate industrial innovation and development over the remainder of this century.

This unique circumstance is created by the emerging *VLSI* technology 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, *VLSI* is a new medium for the realization of computations, it is a statement about system complexity, and not circuit performance. It is a medium with amazing properties, which will profoundly affect and radically change our modes of communication, commerce, education, entertainment, science and the underlying rate of cultural evolution; the quality of human life, in general, can be improved in remarkable ways by these changes. This profound effect

will not be caused by the fact that *VLSI* allows us to make microprocessors, since they are just processors, as we have known them for more than two decades, which hardly exploit *VLSI's* amazing properties. The fundamental property of *VLSI* is that, it is a medium in which computations can be realized that exhibit an unrivalled degree of concurrency. We are not referring to a few cooperating processes, but to a surface of thousands, and in the future possibly millions, of simultaneously active computing elements, which will form modular highly parallel-parallel, possibly heterogeneous, computing structures.

From the historical aspect, the concept of the integrated circuit goes back to the early days of the vacuum tube to valve, when some compound valves were made. However, the first proposal for a solid-state integrated circuit, based on semiconducting material, was made in 1955 by G.W.A. Dummer, at the Royal Research Establishment, Malvern. This concept was brought to practical realization in 1959, simultaneously by Westinghouse Electric Corporation and Texas Instruments, under the U.S. Air Force Molecular Electronics Programme. One can, therefore, compare the dramatic change in the level of integration, which has been made over the past 25 years, and which has brought us from discrete semiconductor devices to *VLSI* components.

The technology has now reached a stage where it is capable of yielding even further improvement in packing density of the circuitry. However, we may face a limitation in our capability of evaluating the circuits themselves in order to guarantee their performance. It is possible that future developments in *VLSI* may be more constrained by computer-aided design and testing, than they will be by the physical

limitations of processing. For a comprehensive chronicle of the evolutionary course of the semiconductor technology, the reader should refer to Larkin [*LARK81*].

In relation with what was said above, approaches to device design 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, *Sticks* languages, *Register Transfer* languages, etc.)[†]. 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*, utilized 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 relative evolution of the component fields comprising microelectronics technology is delineated in *Figure (III.B.1-f1)*[‡].

The actual implementation of such designs requires a highly sophisticated manufacturing technology, found in *silicon wafer* fabrication, the most powerful attribute of 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

[†]*See Ullman [ULLM84], Smith and Dallen [SMIT84].*

[‡]*See Mead [MEAD81], Moralee [MORA82].*

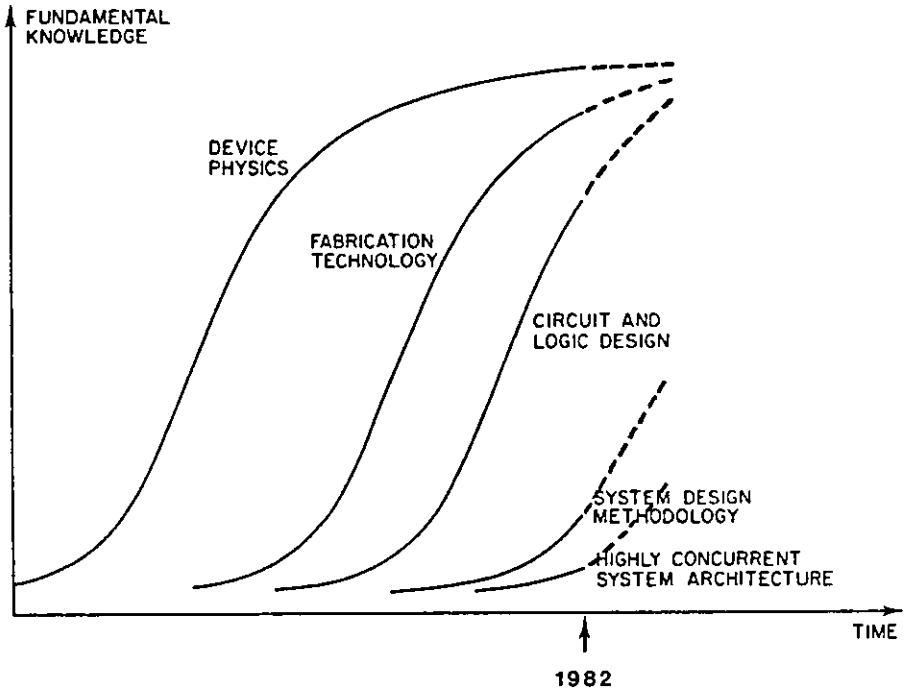


Figure III.B.1-f1: The Relative Evolution of the *'VLSI'* Component Fields.

capabilities. The specification, usually, takes the form of a set of permissible geometries, that may be utilized 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 constraints, 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 [*MEAD80*]).

Although there is not any intention to proceed into any further details of the design rules, we must mention a characteristic and fundamental fact concerning the progressive shrinkage 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-*lambda*. 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 2μm (1 micron(μm) = 10^{-6} meters).

From the aspect of program design, however, to realize them as *VLSI* circuits will be very difficult. This difficulty is, primarily, caused by the many mutual dependencies introduced by concurrency, that is, by the sheer complexity that uncontrolled concurrency inflicts on us. It is undoubtedly an important area of study, but it does not address *VLSI's* most urgent problem requiring mathematical attention, the problem of intellectually mastering the design of ultraconcurrent computing structures.

However, to proceed with planning for the problems even further, in the near, relatively, future the semiconductor industry, under current high-resolution photo-lithographic methods of fabrication, will, probably, reach the fundamental limits after only two or three more doublings in circuit density. Moreover, as these maximum densities are approached, the proportion of manufacturing defects will increase, necessitating costly quality assurance procedures.

In fact, it has recently occurred to researchers, in, both, *Biotechnology* and *Microelectronics*, that these two technologies may be

[†]*Today sub-micron levels are envisaged.*

combined, and this is not an utopia, to allow the utilization of biological materials and processes in the design and fabrication of microelectronic devices. *Bioelectronics*, therefore, a highly revolutionary and far-reaching, in its departure from conventional microelectronics, proposal, will represent a convergence of the requirement in microelectronics for ever-increasing density of circuit elements and the inherent capacity of biological systems to transport electrons on a molecular scale. Its ultimate goal will be the fabrication of a remarkably special electronic device, the *biochip*.

In conclusion, the seeds of the new wave of innovation have, already, begun to take root; undoubtedly, much of the thrust is yet to come, not from the integrated circuit industry itself though, but from the collaborative effort of faculty members in many university computer science departments and scattered individuals throughout the industry.

III.B.2: Fundamental Architectural Concepts In Designing Special-Purpose *VLSI* Computing Structures

High-performance special-purpose *VLSI* computer systems are typically utilized to meet specific application requirements, 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* architecture, a general methodology for mapping high level computations into hardware cellular structures, will be introduced.

The problem of embedding a network of processors and memories, into a set of *VLSI* chips, is currently being examined under a similar spectrum as that for the problem of embedding graphs, whose nodes are computers, or gates, onto grids so as to minimize area. Most of the researchers exploring this problem (e.g. Thompson [*THOM80*], Leiserson [*LEIS80*]), 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 (this is often called *planar*, although, since wires lie in two or more additional layers, non-planar graphs can be embedded), etc.

VLSI chip design rules, typically, specify a grid on which the graph structure of linked gates must be embedded. Many researchers have begun to explore the area and time-complexity of various simple

algorithms, when effected by optimal embeddings of the computation on such a grid.

Although a *VLSI* chip is roughly a 4-8mm square of silicon, yet it outperforms several cubic feet of twenty plus years of old computer components. 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 organization of a chip's circuitry has a very strong effect. In general, regular chip designs make more efficient utilization of silicon area, which is a more natural measurement factor for the circuit size, than the number of transistors. Such designs utilize less area for the wiring amongst transistors, leaving more space for the transistors themselves.

From the memory *capacity* aspect, the number of bits per chip 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 capacity was expanded to 32K bits and it was predicted that by 1985 it would have been increased to 64K bits.

In particular, for the design of special-purpose *VLSI* computer systems, cost-effectiveness has always been a major concern; their fabrication cost must be low enough to justify their specialized, 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 cost of such a system should be relatively small, for it to be more attractive compared to a general-purpose computer, and this can be achieved by the utilization of appropriate architectures. More explanatorily, if the decomposition of a structure into a few types of simple substructures, which are repetively utilized with simple interfaces, is feasible, then significant cost savings can be achieved.

In addition, special-purpose computer systems based on simple and regular designs are likely to be *modular* and, therefore, adjustable to various performance goals - that is, systems cost can be made analogous to the performance required. This fact reveals that accomplishing the architectural challenge for simple and regular designs, yields cost-effective special-purpose computer systems.

Since such *VLSI* computing structures can function as *peripheral devices*, attached to a conventional host computer, typically receiving data and output results, I/O considerations greatly influence the overall performance. 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 system is highly necessary, since it will allow 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 impossible.

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 can 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 chip (or, a module of chips), and modules should be interconnected, so that the whole flow of information through processes is also managed efficiently. These algorithms must support high degrees of concurrency, and employ a simple, regular communication and control, to enable an efficient implementation.

To conclude, since special-purpose *VLSI* 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 (III.B.2-f1)* summarizes the principal stages and tasks' interdependencies in the design of a *VLSI* chip (see Foster and Kung [*FOST80*]). Normally, the algorithm design and specification levels should be independent of the final implementation; however, this is not possible due to the merging of levels in the design, to achieve cost-effective mappings of the high level computations into hardware.

III.B.2.1: The Fundamental Principle, Criteria And Advantages Of *'SYSTOLIC'* Architectures

The choice of an appropriate architecture, for any electronic system, is very closely related to the implementation technology. This is especially true in *VLSI* computing structures whose computational goal

[†] *Since, in practice, problems are typically 'larger' than special-purpose computer systems.*

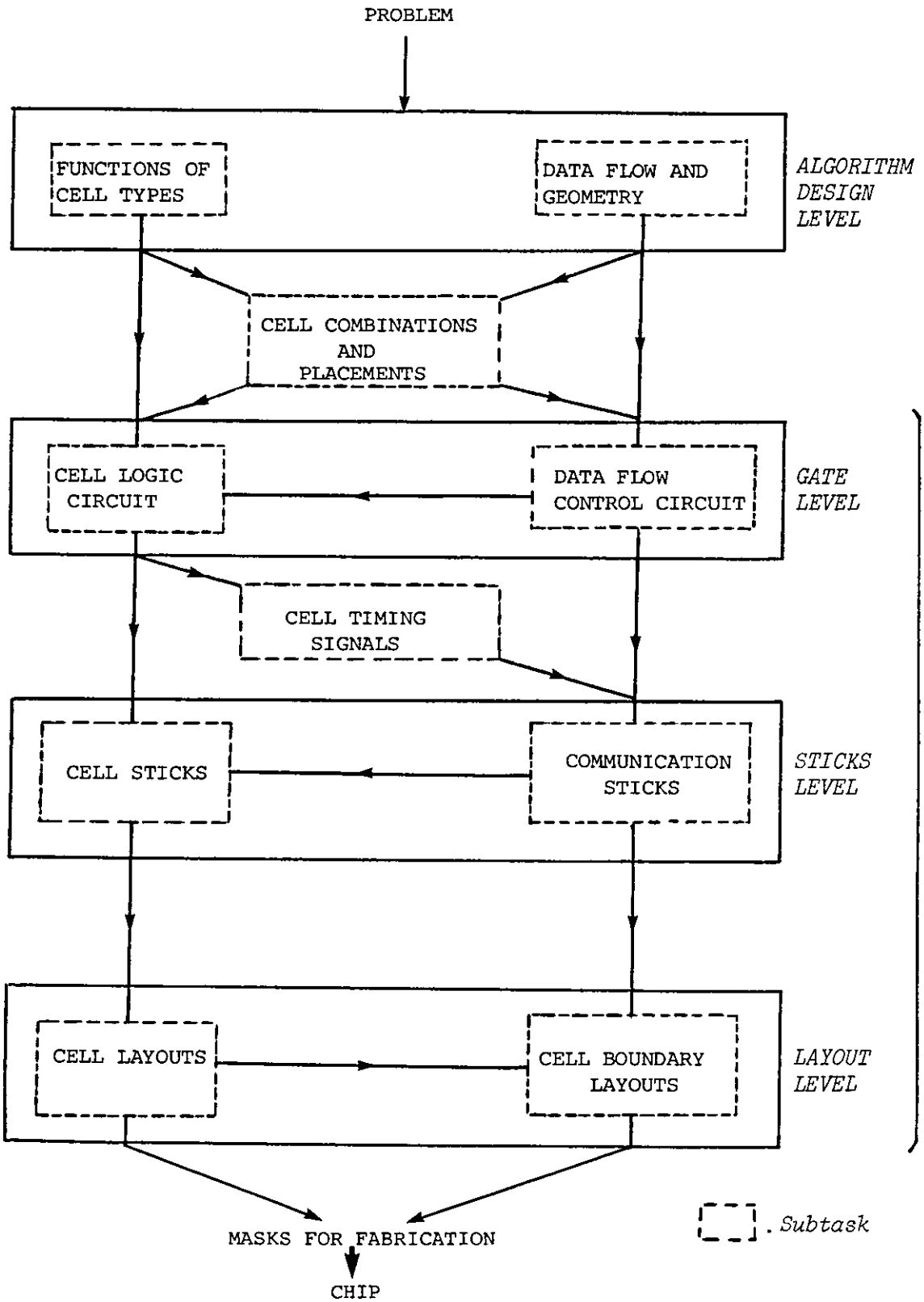


Figure III.B.2.-f1: The Design Stages of a Special-Purpose *VLSI* Chip.

is the implementation of *compute-bound* algorithms, rather than *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 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 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 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*†. Memory bandwidth can be increased by the utilization of either, fast components, which may be quite expensive, or interleaved memories, which may create complicated memory management problems. However, the speed-up of a compute-bound computation may often be accomplished in a, relatively, simple and inexpensive manner; that is, by the *Systolic* architectural approach developed by Kung and his associates at Carnegie-Mellon University. Through this approach, any long distance wiring inside a chip and irregular communication, which can easily dominate the power, chip area and the time required to perform a computation, can be easily prevented. It was, originally, proposed for the *VLSI* implementation of some matrix operations; today, many versions of systolic processors have

†*Since bottlenecks to speeding-up a computation are often due to limited system memory bandwidths, so-called 'von Neumann' bottlenecks, rather than limited processing capabilities.*

been designed by universities and industrial organizations.

A systolic system consists of a set of *cells*,, i.e. 'Processing Elements' - *PEs*, which are regularly interconnected to form a systolic array or a systolic tree. Information in a systolic system flows amongst cells in a pipelined fashion and communication with the environment occurs only at the *'boundary cells'*.

The fundamental principle of a systolic architecture, a systolic array in particular, is illustrated in *Figure (III.B.2.1-f1)*. 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 100ns; then, the conventional memory-processor organization *(i)* has at most a 5MOPS† performance, while, with the same clock rate, the systolic array will result in a possible 30MOPS performance.

Conceptually, the word *Systolic* itself is derived from the physiologists word *'Systole'*, which refers to the rhythmically recurrent contractions of the heart and arteries. More explanatorily, as the heart and arteries pump blood around the human body, a systolic system *pumps* data from memory, around a network of processes representing the computation structure required. In fact, this data stream passes through a number of cells-PEs, on which each process lies, before returning to memory having completed a computation circle.

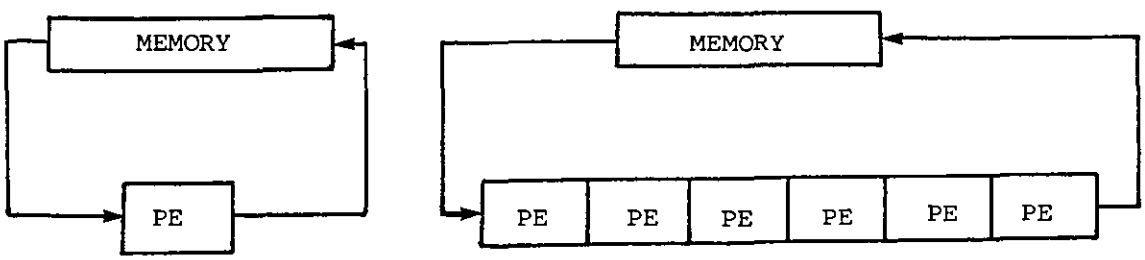
In the body, each organ either takes something from the blood to continue functioning, or acts on it in some predetermined way; in a similar manner, the data flowing around the network can be split into

†*Million Operations Per Second.*

data and control. Each PE can accept control (functioning) signals, or can perform predetermined computations on the data, before the stream is passed onto the next PE, eventually circulating to memory. The crux of this approach is to ensure that once a data item is brought out from the memory, it can be utilized effectively at each cell it passes through, while being *pumped* from cell to cell along the structure.

Sometimes, it is convenient to distinguish the *pumping* mechanism into two distinct states, the *Systole* and the *Diastole* state. The former one, is the actually 'active' *pumping* state in which the communication and controlled data flow occurs amongst PEs; while during the latter, seemingly 'inactive' state, the pumping mechanism recovers and individual PEs perform the due computation on the portion of data stream under their control, setting up the next data and control items to be injected into the stream during the next active state.

Finally, this capability to utilize each input data item a number of times, thus achieving a high computation throughput with only a modest memory bandwidth, is just one of many advantages of the systolic approach. Other, equally significant, criteria and advantages include, modular expansibility, utilization of simple, uniform cells, elimination of global broadcasting, limited fan-in, extensive concurrency and fast response time.

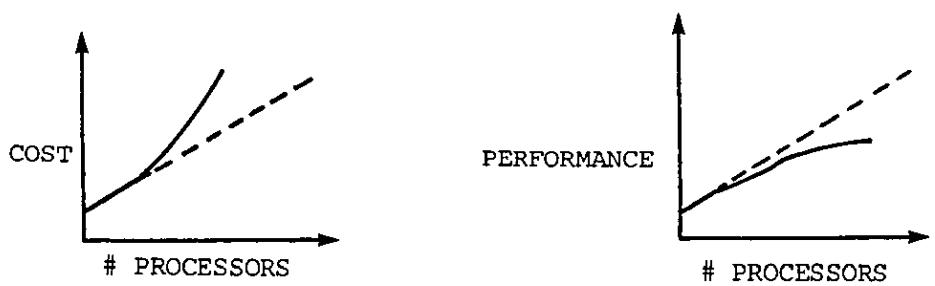


(i) - The *conventional* organization (ii) - A *systolic* processor array

Figure III.B.2.1-f1: The Fundamental Principle of a *'Systolic'* Architecture.

In conclusion, systolic designs based on these criteria meet all the architectural challenges for special-purpose computer systems. A unique characteristic of the systolic approach is that, as the number of cells - PEs expands, the system's cost and performance increase proportionally, provided that the size of the underlying problem is sufficiently large (see *Figure (III.B.2.1-f2,i)*); that is in contrast with other parallel architectures, which are seldom cost-effective for more than a small number of processors (see *Figure (III.B.2.1-f2,ii)*).

(i) - For *systolic* architectures



(ii) - For other *parallel* architectures

Figure III.B.2.1-f2: 'Conceptual' Cost and Performance Curves [*KUNG82*].

III.B.2.2: A Compatibility Taxonomy In The Space Of *'SYSTOLIC'* Computations And *VLSI* Structures

In this conclusive paragraph of *Chapter III* the spatial determination of systolic computations, in the general space of parallel algorithms, is attempted, in relation with appropriate *VLSI* systolic array configurations for their efficient execution; however, any specific algorithm exemplification will be avoided, since this part will be comprehensively exploited later on in the Thesis.

In a parallel algorithm, in general, since more than one task module can be executed at a time, a type of *concurrency control* (i.e. control via shared memory, synchronous centralized control, synchronous or asynchronous distributed control, etc.) is required, which by enforcing desired interactions amongst task modules ensures the correctness of concurrent execution.

The maximal amount of computation that a typical task module of a parallel algorithm can perform, before having to communicate with other modules, is referred to as *module granularity* and ranges from small constants to large computation parts; in other words, the module granularity of a parallel algorithm reflects whether or not the algorithm tends to be communication intensive, since a small module granularity requires analogously frequent intermodule communication. Consequently, in this case, for efficient reasons, it may be desirable to provide proper data paths in hardware, to facilitate such a communication.

Suppose that task modules of a parallel algorithm are connected to represent intermodule communication; then, a geometric layout of

the resulting network is referred to as the *communication geometry* of the algorithm. In *Figure (III.B.2.2-f1)* is presented a classification of the communication geometry of parallel algorithms.

In particular for systolic systems, they correspond to synchronous algorithms that utilize distributed control achieved by simple local control mechanisms and have small constant module granularities. Algorithms that match with systolic systems, utilizing extensive pipelining and multiprocessing, are called *systolic algorithms*. For a low-cost and high-performance chip implementation, it is absolutely crucial that the geometry of the communication paths, in a systolic system, be simple and regular.

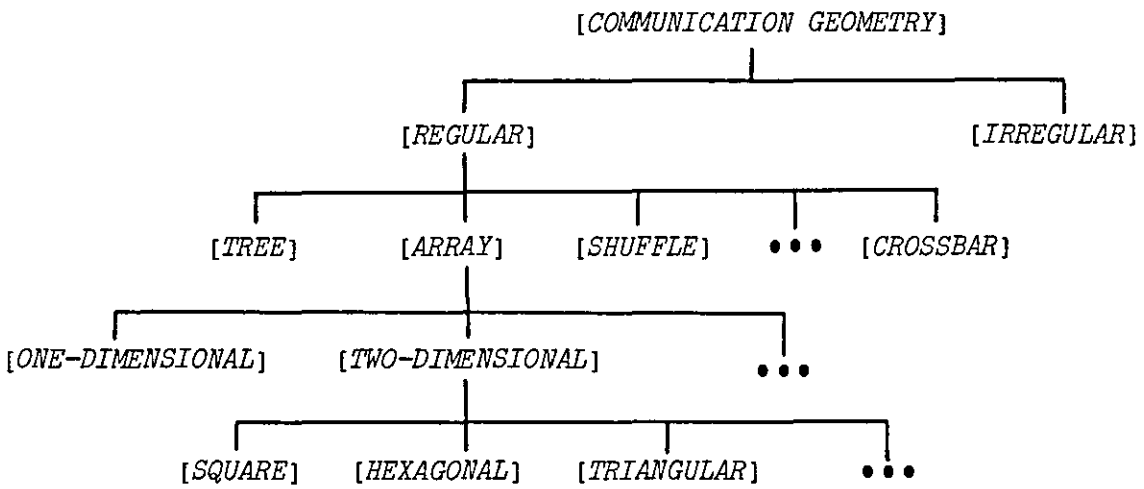


Figure III.B.2.2-f1: A Classification of *Communication Geometry* of Parallel Algorithms.

In summation, the cross product {*concurrency controls*}×{*module granularities*}×{*communication geometries*} represents the *space* of parallel algorithms. One could attempt giving an extensive taxonomy for parallel algorithms, classifying them in terms of their positions

in three-dimensional space, but this space is seen to be large, containing quite a few uninteresting cases; therefore, a reference to a smaller subspace, that nevertheless contains some very significant parallel algorithms, seems to be more realistic. This subspace is the cross product {*systolic*}×{*communication geometries*}, where systolic is a particular position in the area {*concurrency controls*}×{*module granularities*} representing systolic algorithms.

To compare systolic algorithms, in general, with *SIMD* and *MIMD* algorithms, systolic algorithms are the most structured and *MIMD* algorithms are the least structured. Systolic algorithms deal with simple and frequently interacting task modules, while the situation is reversed for *MIMD* algorithms; both concepts will be extensively exemplified in the following *Chapters* of the Thesis. In specific, systolic algorithms are designed for direct hardware implementation, while *MIMD* algorithms are designed for execution on general-purpose Multiprocessors. *SIMD* algorithms, utilizing a central control, are found lying between these two classes of algorithms.

VLSI systolic arrays can assume many different structures for different compute-bound algorithms. *Figure (III.B.2.2-f2)* exhibits a variety of systolic array configurations. Their potential utilization for various problem cases is summarized in *Table (III.B.2.2-t1)*.

Finally, in a particular historical retrospection, the *2-D*[†] *square array* is perhaps one of the first communication geometries studied by researchers interested in parallel processing; work in *cellular automata*, concerned with computations distributed in a *2-D* orthogonally

[†] *Dimensional.*

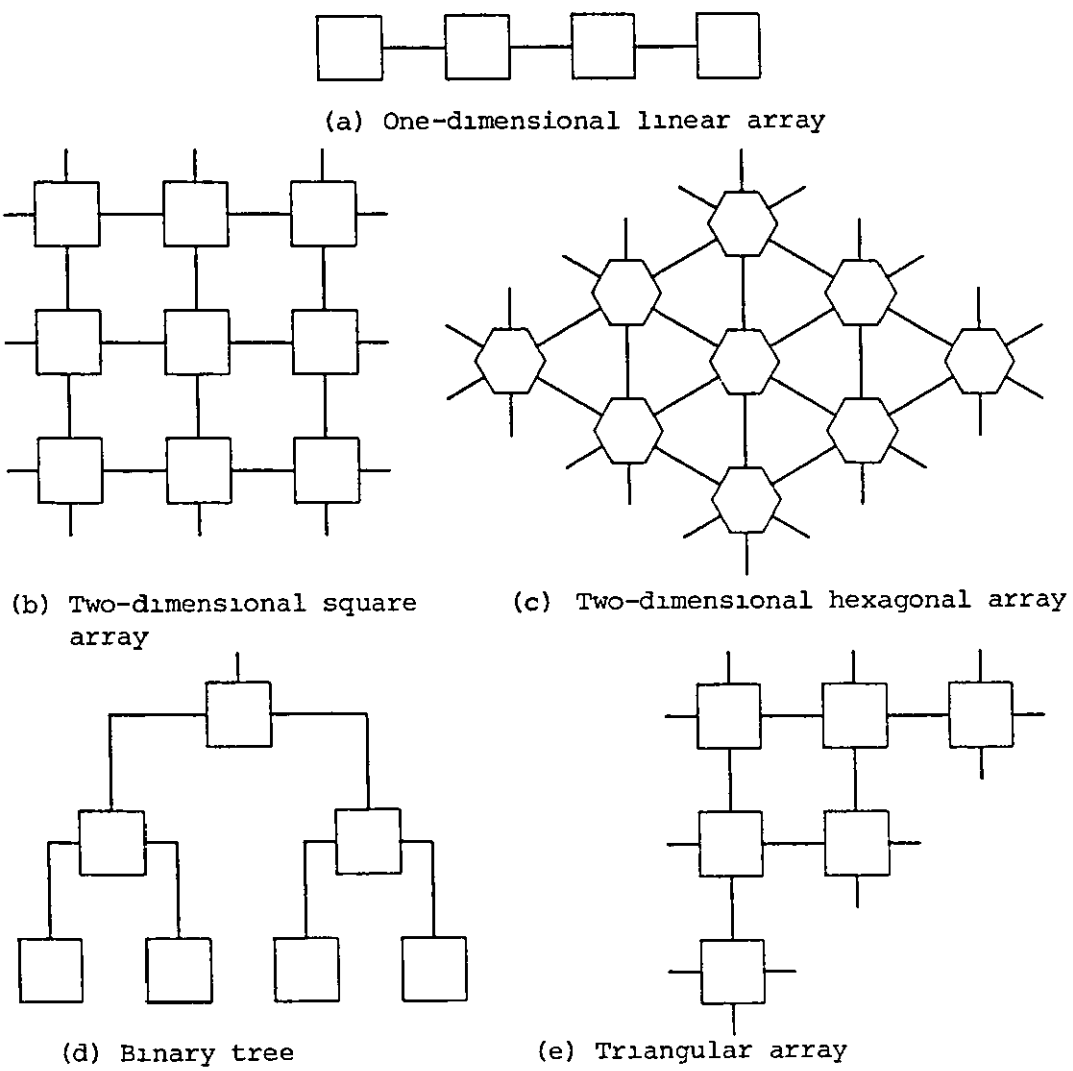


Figure III.B.2.2-f2: Various *'Systolic'* Array Configurations.

'SYSTOLIC' PROCESSOR ARRAY STRUCTURE	PROBLEM CASES
1-D linear arrays	: FIR-filter, convolution, 'Discrete Fourier Transform' - *DFT* , 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.
2-D square arrays	: Dynamic programming for optimal parenthesization, image processing, pattern matching, numerical relaxation, graph algorithms involving adjacency matrices.
2-D hexagonal arrays	: Matrix problems (matrix multiplication, *LU*-decomposition by Gaussian elimination without pivoting, *QR*-factorization), transitive closure, relational database operations, *DFT*.
Trees	: Searching algorithms (queries on nearest neighbour, rank, etc., systolic search tree), recurrence evaluation.
Triangular arrays	: Inversion of triangular matrix, formal language recognition.

Table III.B.2.2-t1: The Potential Utilization of *'Systolic'* Array Configurations.

connected array, was initiated by von Neumann [*VONN66*], in 1966. More recently, because of the technological advancement, there has been an increasing interest in designing algorithms for cellular arrays. In specific for pattern recognition algorithms, the pattern matching chip described in Foster and Kung [*FOST80*] has recently been fabricated, tested, and found to work.

To conclude, the main advantage provided by the *tree* structure, which is not shared by any array structure, is the logarithmic-time property for broadcasting, searching, and fan-in. However, if the majority of communications are not confined to processors at low levels, then the processors at high levels may become bottlenecks causing a certain drawback. Algorithms that can take advantage of the potentiality provided by the tree structure, while avoiding its possible drawback, exhibit a significant interest.

To this end, we have completed a brief and taxonomically disciplined *state-of-the-art* survey, with as much (as was possible) up-to-date information on the *parallel* computing environment. Certainly, in the space of the implemented computer systems a plethora of them, especially some currently, at the time of completing this *Section*, available commercial Multiprocessor systems (e.g. *Cray X-MP*, Denelcor's 'Heterogeneous Element Processor' - *HEP*, possibly *Cray 2*, etc.) have been omitted; but, this was logically bound to happen with the tremendously rapid technological advancement.

Furthermore, we have not directly dealt with networks of autonomous computers connected via communication lines, such as the *ETHERNET*, *ARPANET*, *Cambridge Ring*, etc., which have been successfully

established and have been operating for a number of years. The principal reason was that, these computer complexes are conceptually much more *indirectly* or *loosely coupled*† compared to the *MIMD* designs we have introduced; in fact, their Operating Systems are usually built and tuned to cope with electronic mail and document handling (e.g. sharing of resources, editing, displaying, printing, etc.), rather than the efficient execution of a single program by many processors.

In particular for the contents of the present *Section*, they will be more analytically complemented in later *Chapters* of the Thesis, which will deal with special algorithms for direct hardware implementation.

†*Hence, out of the scope of this Thesis.*

CHAPTER IV: AN INVESTIGATION ON THE POTENTIAL PARALLELISM OF A NEW CLASS OF GROUP EXPLICIT METHODS FOR THE SOLUTION OF PARABOLIC PARTIAL DIFFERENTIAL EQUATIONS

- *SECTION A:* FUNDAMENTAL CONCEPTS OF DIFFERENTIAL EQUATIONS
  - *IV.A.1:* INTRODUCTORY REMARKS
  - *IV.A.2:* MATHEMATICAL PRELIMINARIES OF DIFFERENTIAL EQUATIONS
  - *IV.A.3:* CANONICAL CLASSIFICATION OF PARTIAL DIFFERENTIAL EQUATIONS
    - *IV.A.3.1:* BOUNDARY CONDITIONS
    - *IV.A.3.2:* MATHEMATICAL PHYSICS AND WELL-POSEDNESS OF PROBLEMS
    - *IV.A.3.3:* ANALYTICAL/NUMERICAL APPROXIMATE METHODS OF SOLUTION
- *SECTION B:* PARALLEL EXPLOITATION OF EXPLICIT METHODS FOR THE SOLUTION OF NON-LINEAR PARABOLIC EQUATIONS
  - *IV.B.1:* PRELIMINARY CONCEPTS AND NOTATIONS OF FINITE-DIFFERENCE APPROXIMATIONS TO DERIVATIVES
  - *IV.B.2:* VARIOUS FINITE-DIFFERENCE APPROXIMATION SCHEMES TO A NON-LINEAR PARABOLIC PROBLEM
    - *IV.B.1.1:* PARABOLIC EQUATIONS IN ONE SPACE DIMENSION AND DISCRETIZING FINITE-DIFFERENCE FORMULAE
    - *IV.B.1.2:* A DESCRIPTIVE TREATMENT OF THE CONVERGENCE, STABILITY, AND CONSISTENCY OR COMPATIBILITY CONCEPTS
  - *IV.B.3:* THE NEW CLASS OF 'GROUP EXPLICIT' - *GE* SOLUTION METHODS
    - *IV.B.3.1:* THE STANDARD EXPLICIT METHOD: PERFORMANCE MODEL, EXPERIMENTAL RESULTS AND PERFORMANCE ANALYSIS ON THE '*NEPTUNE*' PROTOTYPE SYSTEM
    - *IV.B.3.2:* THE 'GROUP EXPLICIT WITH UNGROUPED ENDS' - *GEU* METHOD: EXPERIMENTAL RESULTS AND PERFORMANCE ANALYSIS ON THE '*NEPTUNE*' PROTOTYPE SYSTEM
    - *IV.B.3.3:* THE 'GROUP EXPLICIT COMPLETE' - *GEC* METHOD: EXPERIMENTAL RESULTS AND PERFORMANCE ANALYSIS ON THE '*NEPTUNE*' PROTOTYPE SYSTEM
    - *IV.B.3.4:* THE '(SINGLE) ALTERNATING GROUP EXPLICIT' - *(S)AGE* METHOD: EXPERIMENTAL RESULTS AND PERFORMANCE ANALYSIS ON THE '*NEPTUNE*' PROTOTYPE SYSTEM
    - *IV.B.3.5:* THE '(DOUBLE) ALTERNATING GROUP EXPLICIT' - *(D)AGE* METHOD: EXPERIMENTAL RESULTS AND PERFORMANCE ANALYSIS ON THE '*NEPTUNE*' PROTOTYPE SYSTEM
    - *IV.B.3.6:* A '(MODIFIED DOUBLE) ALTERNATING GROUP EXPLICIT' - *(MD)AGE* METHOD: EXPERIMENTAL RESULTS AND PERFORMANCE ANALYSIS ON THE '*NEPTUNE*' PROTOTYPE SYSTEM
    - *IV.B.3.7:* INDICATIVE EXPERIMENTAL RESULTS AND PERFORMANCE MEASUREMENTS OF THE *GEU* METHOD ON SIMD AND PIPELINED VECTOR COMPUTERS
  - *IV.B.4:* RELATIVE PERFORMANCE COMPARISONS AND CONCLUSIVE REMARKS ON THE *GE* METHODS

Chapter IV

An Investigation On The Potential Parallelism Of A New Class Of Group Explicit Methods For The Solution Of Parabolic Partial Differential Equations

Section Å

Fundamental Concepts Of Differential Equations

IV.A.1: INTRODUCTORY REMARKS

In this *Chapter* we investigate the applicability of a new class of Group Explicit methods, for the numerical solution of parabolic partial differential equations, to parallel processing.

In the present *Section A* we initiate the reader to the necessary preliminary mathematical background of differential equations and in particular the class of partial differential equations reviewing all the related notations, conditions and concepts essential for their proper use.

In *Section B* we confine ourselves to finite-difference methods only, as applied to solve parabolic partial differential equations, discussing again some of their fundamental concepts and notations, while a descriptive treatment of the convergence, stability and consistency or compatibility concepts is given. The *Section* continues by presenting the new class of powerful solution methods, the Group Explicit methods, which use stable asymmetric approximations to the partial differential equation coupled in groups of 2 adjacent points (4 for two dimensions) on the grid; this results in implicit equations

which can be easily converted to explicit form and which offer many advantages especially for use on parallel computers. Furthermore, by judicious use of alternating this strategy on the grid points of the domain results in new explicit algorithms which possess unconditional stability. The merit of this approach also results in more accurate solutions because of truncation error cancellations.

The potential parallelism of these methods in comparison with the Standard Explicit method is extensively exploited, the experimental vehicle being Burgers' non-linear parabolic partial differential equation of second-order.

For the performance analysis of all parallel implementations in the Thesis, a detailed performance model is established along with all the particular general formulae for the evaluation of its various parameters. The principle behind this analysis is that parallel processing involves the sharing of some resources which have a limited availability. This has the consequence that there is a limit to the number of demands that can be satisfied and some of them must wait if there are some competing ones. These demands are determined by the program, while the availability and allocation algorithm are properties of the system. Therefore, the performance of a given algorithm on a given parallel architecture can be obtained by analyzing the properties of the system under various theoretical demand patterns. This analysis is substantiated by an analysis of the resources provided by the *NEPTUNE* prototype system and the resources demanded by the investigated parallel algorithms. In particular, all different implementations are analyzed in a theoretical (i.e. program dependent) and experimental (i.e. system dependent) manner for cross-verification purposes.

Finally, some indicative experimental results and performance measurements of the Group Explicit methods, again in comparison with the Standard Explicit method, are presented, obtained on *SIMD* and *Pipelined Vector* computers, thus completing our study of the potential and suitability of these methods for parallel processing. The *Chapter* concludes with some relative performance comparisons and general remarks for future investigation on the Group Explicit methods.

IV.A.2: MATHEMATICAL PRELIMINARIES OF DIFFERENTIAL EQUATIONS

The process of gradual and continuous growth or increase may be observed in innumerable instances, what is of real importance though, is not necessarily the actual amount of growth or increase, but the *rate of growth or increase*. It is this problem, closely connected with infinitesimal increases, that is the basis of the Infinitesimal Calculus, and more especially that part of it which is called the *Differential Calculus*.

As a brief historical note, the word 'calculus', linguistically, is the Latin name for a stone which was employed by the Romans for reckoning, i.e. for 'calculation'. The calculus is one of the most powerful mathematical inventions whose discovery credit has been claimed for both, Sir Isaac Newton and Leibnitz, the great German mathematician.

To become more deterministic, the study and use of *Differential Equations* arises from the need to express and subsequently solve a variety of physical phenomena and problems, mainly, in Physics and Engineering science. In actual fact, they are concerned with the rates of change of unknown quantities, called *dependent variables*, such as,temperature, pressure, etc., with respect to one (or more) *independent variable* representing length, angle, etc. The majority of the above problems fall naturally into one of three *physical* categories: *Equilibrium* problems, *Eigenvalue* problems and *Propagation* problems.

The problems of the *first* category are problems of *steady-state* (i.e. time-invariant), in which the equilibrium configuration, in a given domain D,[†] is to be determined by solving a specific differential

[†] *Very often, but not always, the integration domain D is 'closed' and 'bounded' (see Ames [AMES69], p.3).*

equation within this domain, subject to certain boundary conditions on the boundary of the domain. In mathematical terminology such problems are known as *boundary-value problems*. Typical physical examples include, steady viscous flow, steady temperature distributions, etc.

The *Eigenvalue* problems may be thought of as an extension of the equilibrium problems wherein 'critical values' of certain parameters are to be determined, in addition to the corresponding steady-state configurations. Typical physical examples include, buckling and stability of structures, natural frequency problems in vibrations, etc.

The *Propagation* problems are *initial-value problems* that have an unsteady state or transient nature; in other terms, they are concerned with the prediction of the subsequent behaviour of a system given the initial state. In mathematical parlance such problems are known as *initial/boundary-value problems*[†]. Typical physical examples include, the propagation of pressure waves in fluid, propagation of heat, etc. The distinction between equilibrium and propagation problems lies in the fact that, in the former the entire solution is passed on by a *jury* requiring satisfaction of all the boundary conditions and internal requirements; while in propagation problems the solution *marches* out from the initial state guided and modified in transit by the side boundary conditions (see Richardson [*RICH25*]).

A *Differential Equation* can be defined as an equation which involves *derivatives*. If there is only a single independent variable, then the derivatives are 'ordinary' derivatives and the equations are called *Ordinary Differential Equations*. However, in most cases, the

[†]*Sometimes only the terminology 'initial-value problem' is utilized (see Ames [AMES69], pp.3-5).*

dependent variable, in any of the categories of problems mentioned earlier, is expressed in terms of several independent variables. Such problems inherently give rise to the need for 'partial' derivatives in the description of their behaviour. The study of differential equations arising from these problems constitutes the field of *Partial Differential Equations*.

The *Order* of a differential equation is the order of the highest derivative which occurs, while the *Degree* of a differential equation, which can be written as a polynomial in the derivatives, is the degree of the highest ordered derivative which the equation contains.

The general mathematical form of an *ordinary differential equation* (henceforth abbreviated as *o.d.e.*), for a dependent variable $u(x)$, is a relation such as,

$$F(x,u,u',\ldots,u^{(r-1)},u^{(r)},\ldots) = 0, \ (u^{(r)}=\frac{d^r u}{dx^r} \ , \text{ for } r\geq 1), \qquad \textit{(IV.A.2:1)}$$

where F is a given function of the independent variable x, the 'unknown' function u and a *finite* number of the latter's derivatives.

From the linearity aspect, any o.d.e. of order n determined by a form such as,

$$p_0 \frac{d^n u}{dx^n} + p_1 \frac{d^{n-1} u}{dx^{n-1}} + p_2 \frac{d^{n-2} u}{dx^{n-2}} + \ldots + p_{n-1} \frac{du}{dx} + p_n u = h, \qquad \textit{(IV.A.2:2)}$$

where $p_0 \neq 0, p_1, p_2, \ldots, p_n, h$ are functions of x or constants, is said to be *linear*; any other form of equations is considered to be *non-linear*, i.e. if at least one of the terms involved with the dependent variable or its derivatives is not of the same degree as the other terms. In particular, if $h=0$, then *(IV.A.2:2)* takes the form,

$$p_0 \frac{d^n u}{dx^n} + p_1 \frac{d^{n-1} u}{dx^{n-1}} + p_2 \frac{d^{n-2} u}{dx^{n-2}} + \ldots + p_{n-1} \frac{du}{dx} + p_n u = 0, \qquad (IV.A.2:3)$$

and is additionally called *homogeneous* to indicate that *all* of the terms are of the same (i.e. *first*) degree in u and its derivatives.

The general mathematical form of a *partial differential equation* (henceforth abbreviated as *p.d.e.*), for a dependent variable $u(x,y,\ldots)$, is a relation such as,

$$F(x,y,\ldots,u,\frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \ldots, \frac{\partial^2 u}{\partial x^2}, \frac{\partial^2 u}{\partial y^2}, \ldots, \frac{\partial^2 u}{\partial x \partial y}, \ldots)^{\dagger} = 0, \qquad (IV.A.2:4)$$

where F is a given function of the independent variables $x,y,\ldots$, the 'unknown' function u and a *finite* number of the latter's partial derivatives. The independent variables $x,y,\ldots$ are real (unless if stated otherwise) and u and its derivatives occurring in *(IV.A.2:4)* are *continuous* functions of $x,y,\ldots$ in some real domain D, in the space of these independent variables.

In a similar manner, a p.d.e. of the above *(IV.A.2:4)* form is said to be *linear* if F is linear in the unknown function u and all its partial derivatives; whilst, it is said to be *quasi-linear* if F is linear in the highest order derivatives and the coefficients of F depend not only on the independent variables, but also on $u, \partial u/\partial x, \partial u/\partial y, \ldots,$ etc. A linear equation is a special case of a quasi-linear equation. For example, the equation

$$\sqrt{x} \frac{\partial u}{\partial x} + u \frac{\partial u}{\partial y} = -u^2 \qquad (IV.A.2:5)$$

is a first-order *quasi-linear* p.d.e., and the equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - 32u = 0 \qquad (IV.A.2:6)$$

---

†*Various authors often use the conventional notations:* $u_x = \partial u/\partial x$, $u_{xx} = \partial^2 u/\partial x^2, \ldots,$ *etc.*

is a second-order *linear* p.d.e. Meanwhile, the equation

$$\left(\frac{\partial^2 u}{\partial x^2}\right)\left(\frac{\partial^2 u}{\partial y^2}\right) - \left(\frac{\partial^2 u}{\partial x \partial y}\right)^2 = 0 \qquad (IV.A.2:7)$$

is an example of a second-order *non-linear* p.d.e.

In these examples x and y are the independent variables, while $u=u(x,y)$ is the dependent variable whose form is to be determined.

On the other hand, the definition of the *homogeneous* attribute for a linear p.d.e. is slightly vague, because of the fact that there is not any agreement amongst authors in the use of this term. So, we may come across a case that a linear p.d.e. such as,

$$x^2 \frac{\partial^3 u}{\partial x^3} + xy \frac{\partial^3 u}{\partial x^2 \partial y} + 2 \frac{\partial^3 u}{\partial x \partial y^2} + \frac{\partial^3 u}{\partial y^3} = x^2 + y^3 \ , \qquad (IV.A.2:8)$$

in which the derivatives involved are all of the same order, is considered to be *homogeneous*; however, a more widely acceptable definition is that, a linear p.d.e. is considered to be *homogeneous* if each term contains either, the dependent variable, or one of its derivatives. For example, the equation

$$\frac{\partial u}{\partial t} - \sigma \frac{\partial^2 u}{\partial x^2} = 0 \quad (\textit{heat-conduction equation}) \qquad (IV.A.2:9)$$

is homogeneous, while, according to the latter definition, the equation

$$\frac{\partial u}{\partial t} - a \frac{\partial^2 u}{\partial x^2} = f(x,t) \ , \ a>0 \qquad (IV.A.2:10)$$

where $f(x,t)$ is a given function, is an *inhomogeneous* equation.

The problem of finding a *solution* (or an *integral*) of a general elementary differential equation is essentially that of recovering the *primitive* which gives rise to the differential equation. In other words, the problem of solving a differential equation of order n is

in fact that of finding a *relation* amongst the variables involving n independent, and *essential*[†], arbitrary constants, which together with the derivatives obtained from it satisfy the differential equation. The conditions under which one can be assured that a differential equation is solvable are given by *Existence Theorems*.

A *particular solution* of a differential equation is one obtained from the primitive by assigning definite values to the arbitrary constants. In geometrical terms, the primitive is the equation of a family of curves and a particular solution is the equation of one of the curves. These curves are called *integral curves* of the differential equation.

However, we must note that a given form of the primitive may not include all of the particular solutions; moreover, a differential equation may have solutions which cannot be obtained from the primitive by any manipulation of the arbitrary constants. Such solutions are called *singular solutions*. The primitive of a differential equation is usually called *the general solution* of the equation, although, due to the above remarks, some authors consider it as *a general solution* of the equation.

In specific for the field of p.d.e.'s, the problem of finding a solution to a p.d.e. is a very difficult problem due to the lack of a general method of solution, except for certain special types of linear or quasi-linear equations. In any case, however, a function u defined in some region Ω, in the space of the independent variables, will be called a *classical solution* of *(IV.A.2:4)* in the region Ω, if throughout this region the function u has continuous partial

[†]*Namely, they cannot be replaced by a smaller number of constants.*

derivatives up to the order of the equation inclusively and substitution of $u(x,y,\ldots)$ into the original equation reduces that equation to an identity. The requirement that the first m (i.e. as the order of the p.d.e.) partial derivatives exist is often unjustified from a physical, and sometimes even from a mathematical, point of view.

Therefore, in addition to the concept of a classical solution, the concept of a *generalized solution* of a p.d.e. was introduced by S.L. Sobolev (see Mikhlin [*MIKH67*]). However, a generalized solution is usually of very little use, since it has to satisfy certain boundary conditions arising from the nature of the problem itself. The simplest formal definition of such a solution stands as: 'If there exists a sequence of classical solutions of the given differential equation in a region Ω and if this sequence converges uniformly to some function u in an arbitrary subregion in the interior of the region Ω, this function u is said to be a *generalized solution* of the given differential equation in the region Ω'.

Finally, similar to an o.d.e., if $u_1,u_2,\ldots,u_n$ are n different solutions of a linear homogeneous p.d.e. in some given domain, then,

$$u = c_1u_1 + c_2u_2 + \ldots + c_nu_n \qquad (IV.A.2:11)$$

is also a solution in the same domain, where $c_1,c_2,\ldots,c_n$ are arbitrary constants.

IV.A.3: CANONICAL CLASSIFICATION OF PARTIAL DIFFERENTIAL EQUATIONS

The physical classification of problems, presented in the previous paragraph, briefly emphasized on their distinctive features, a fact which, in addition, suggests that their governing equations are quite different in nature as well. However, not all p.d.e.'s can be classified into well-defined categories. The most comprehensive analysis relates to linear or quasi-linear equations of the second-order in two independent variables (see Vichnevetsky [*VICH81*],p.8).

The classification of p.d.e.'s, in a descriptive-like manner, depends on the type of the physical problems to which they apply, e.g. the *heat-conduction equation*, the *wave equation*, etc.; normally, though, it is best accomplished by developing the concept of *characteristics*.

In a more analytic presentation of the latter concept, let us consider the second-order quasi-linear p.d.e.

$$a \frac{\partial^2 u}{\partial x^2} + b \frac{\partial^2 u}{\partial x \partial y} + c \frac{\partial^2 u}{\partial y^2} + e = 0 , \qquad (IV.A.3:1)$$

where a,b,c and e may be functions of $x,y,u,\partial u/\partial x$ and $\partial u/\partial y$, but not of $\partial^2 u/\partial x^2, \partial^2 u/\partial x \partial y$ and $\partial^2 u/\partial y^2$, i.e. the second-order derivatives occur only to the first degree. It will be shown that *at every point* of the x-y plane there are two directions in which the integration of the p.d.e. reduces to the integration of an equation involving total differentials only; in other words, in these directions the equation to be integrated is not complicated by the presence of partial derivatives in other directions.

Let us consider the following denotation for the first- and second-order derivatives,

$$\frac{\partial u}{\partial x} = p; \ \frac{\partial u}{\partial y} = q; \ \frac{\partial^2 u}{\partial x^2} = r; \ \frac{\partial^2 u}{\partial x \partial y} = s, \text{and } \frac{\partial^2 u}{\partial y^2} = t.$$

Let $C^{\dagger}$ be a curve on the x-y plane on which the values of u and its derivatives above satisfy equation *(IV.A.3:1)*. Therefore, the differentials of p and q in directions tangential to C satisfy the equations,

$$dp = \frac{\partial p}{\partial x}\,dx + \frac{\partial p}{\partial y}\,dy = r\,dx + s\,dy \qquad (IV.A.3.2)$$

and

$$dq = \frac{\partial q}{\partial x}\,dx + \frac{\partial q}{\partial y}\,dy = s\,dx + t\,dy\ , \qquad (IV.A.3:3)$$

where

$$ar + bs + ct + e = 0 \qquad (IV.A.3:4)$$

and dy/dx is the slope of the tangent to C at point $P(x,y)$.

Elimination of r and t from *(IV.A.3:4)* using *(IV.A.3:2)* and *(IV.A.3:3)* results in

$$\frac{a}{dx}\,(dp - s\,dy) + bs + \frac{c}{dy}\,(dq - s\,dx) + e = 0$$

i.e.,

$$s\left\{a\left(\frac{dy}{dx}\right)^2 - b\left(\frac{dy}{dx}\right) + c\right\} - \left\{a\frac{dp}{dx}\frac{dy}{dx} + c\,\frac{dq}{dx} + e\,\frac{dy}{dx}\right\} = 0. \qquad (IV.A.3:5)$$

Now, by choosing the curve C so that the slope of the tangent, at every point on it, is a root of the equation

$$a\left(\frac{dy}{dx}\right)^2 - b\left(\frac{dy}{dx}\right) + c = 0\ , \qquad (IV.A.3:6)$$

s is also eliminated.

Therefore, *(IV.A.3:5)* leads to

$$a\,\frac{dp}{dx}\frac{dy}{dx} + c\,\frac{dq}{dx} + e\,\frac{dy}{dx} = 0\ . \qquad (IV.A.3:7)$$

Consequently, it is apparent that *at every point* $P(x,y)$ of the solution domain there are two directions, given by the roots of equation *(IV.A.3:6)*, along which there is a relationship, given by equation *(IV.A.3:7)*, between the total differentials dp and dq

---

†*In fact, this is not a curve on which the initial-values of* u,p *and* q *are given.*

with respect to x and y. The directions given by the roots of equation *(IV.A.3:6)* are called the *characteristic directions* and the p.d.e. is considered to be *hyperbolic*, *parabolic* or *elliptic* according to whether these roots are real and distinct, equal, or complex, respectively, i.e. according to whether $b^2-4ac \gtreqless 0$.

The reader must bear in mind that the classification of a p.d.e., and therefore its solution method, may depend on the region of the x-y plane under consideration; namely, it is possible for a p.d.e. to change classification within different regions of the same domain wherein the problem is defined. For example, the characteristic directions of the equation

$$y\frac{\partial^2 u}{\partial x^2} + x\frac{\partial^2 u}{\partial x \partial y} + y\frac{\partial^2 u}{\partial y^2} = F(x,y,u,p,q) \qquad \textit{(IV.A.3:8)}$$

are given by the roots m_1, m_2 of the quadratic

$$ym^2 - xm + y = 0 \ , \quad m = dy/dx \ , \qquad \textit{(IV.A.3:9)}$$

which are real, equal or complex according to $x^2-4y^2 \gtreqless 0$. Consequently, the equation is *hyperbolic* for $|x|>2|y|$, *parabolic* along $|x|=2|y|$ and *elliptic* for $|x|<2|y|$. However, such cases are infrequent and therefore can be ignored with little loss of generality (see Vichnevetsky [*VICH81*], p.10).

In general, the best known and most suitable representatives of these classes[†] of p.d.e.'s are,
the hyperbolic *wave equation*

$$\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2} \ , \text{ (c is a real constant)} \qquad \textit{(IV.A.3:10)}$$

[†] *Their general 'canonical' forms, in two independent variables, are:*

$$u_{xx}-u_{tt}+\ldots = 0 \text{ (in the hyperbolic case)}$$
$$u_{tt}+\ldots = 0 \text{ (in the parabolic case)}$$
$$u_{xx}+u_{yy}+\ldots = 0 \text{ (in the elliptic case).}$$

the parabolic *heat-conduction* or *diffusion equation*

$$\frac{\partial u}{\partial t} = \sigma \frac{\partial^2 u}{\partial x^2} \quad , \ (\sigma \text{ is a real constant}) \qquad (IV.A.3{:}11)$$

and the elliptic

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \begin{cases} -g(x,y) & \textit{Poisson's equation} \\ \textit{or} & \\ 0 & \textit{Laplace's equation.} \end{cases} \qquad (IV.A.3{:}12)$$

In concern with the governing equations of the physical classification presented in the previous paragraph, the original mathematical formulation of an *equilibrium* problem will generate an *elliptic* equation or system, albeit, possibly, later mathematical approximations may alter the type. In some cases the curve along which the information is given may be *closed*, while for others may be not; for example, the general (*analytical*) solution of a two-dimensional elliptic equation is a function of the space coordinates x and y, which not only satisfies the p.d.e. *at every point* $P_{i,j}$ of the area S,[†] inside a plane *closed* curve C, but also satisfies certain boundary conditions *at every point* on this boundary curve C [see *Figure (IV.A.3-f1)*].

On the other hand, *propagation*[‡] problems are governed by *parabolic* or *hyperbolic* equations, which can give rise to an *open-ended* area of integration S into which the solution propagates. The initial and boundary conditions are known and are, normally, located along the x-axis and along parallel lines perpendicular to the x-axis, respectively, as shown in *Figure (IV.A.3-f2)*. In a similar way, the solution $u(x,t)$ must fulfil the given conditions along curve C and satisfy the p.d.e. *at every point* $P_{i,j}$ of the infinite area of integration S bounded by the *open* curve C on the x-t plane.

[†] *Called the area of integration.*

[‡] *Or, in general, problems involving time - 't' as one independent variable.*

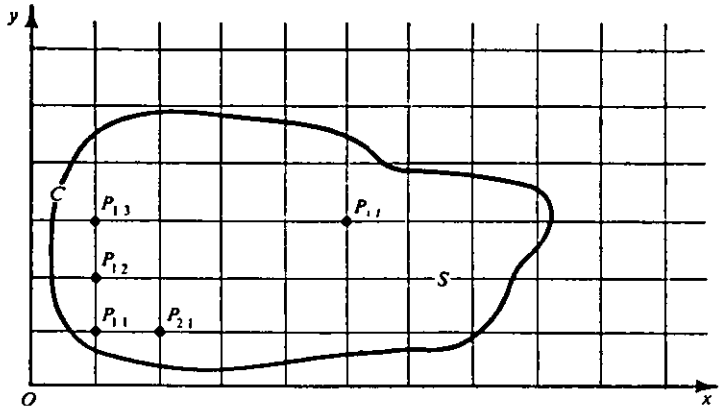


Figure IV.A.3-f1: The Area of Integration S and the Boundary Curve C.

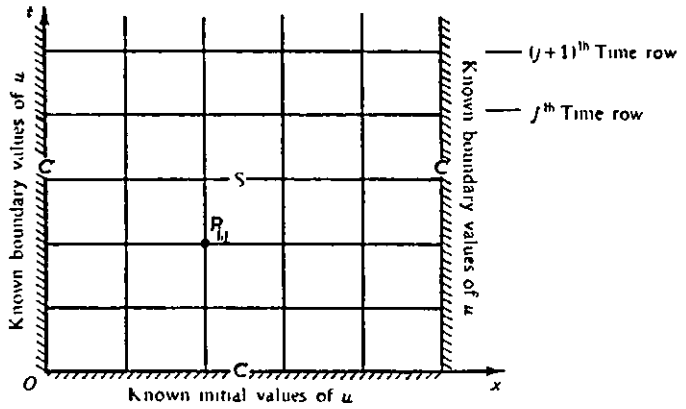


Figure IV.A.3-f2: The *Open-Ended*, Area of Integration S and Curve C.

Finally, the generalization of the name of hyperbolic, parabolic and elliptic to equations that are neither second-order, nor in two independent variables, but which possess similar properties, is often done as a matter of fact. The conclusion is that the majority of problems of practical importance can be related to these three *canonical* classes, although, sometimes, a more complicated[†] classification is carried out.

[†]*For example, a p.d.e. of elliptic type, in three independent variables, is referred to with a three-dimensional analogous name, such as, 'ellipsoidal' p.d.e.*

IV.A.3.1: BOUNDARY CONDITIONS

The solution of a p.d.e., as was mentioned earlier in this *Chapter*, has to satisfy, in particular, some *boundary conditions* arising from the formulation of the problem itself. In accordance with the specific type of the occurred boundary conditions *four* different basic categories of problems can be distinguished, which arise frequently in the description of various physical phenomena. These are:

(i) - The First Boundary-Value Problem (the Dirichlet Problem), where a solution u is sought, which is a continuous function within a specified region and satisfies the given values

$$u\big|_C = \phi \qquad (IV.A.3.1:1)$$

on the boundary C.

(ii) - The Second Boundary-Value Problem (the Neumann Problem), where a solution u is sought as before, which has to satisfy the normal derivatives

$$\left.\frac{\partial u}{\partial n}\right|_C = \psi \qquad (IV.A.3.1:2)$$

on the boundary C of that region.

(iii) - The Third Boundary-Value Problem (Mixed or Robbin's Problem), where a solution u is sought, which is again a continuous function within a specified region and satisfies a combination of u and its derivatives, namely,

$$\left[\frac{\partial u}{\partial n} + hu\right]_C = \psi \qquad (IV.A.3.1:3)$$

on the boundary C.

All of the above problems are said to be *homogeneous* if ϕ,ψ are equal to zero.

The problem of the steady-state temperature distribution, in particular, can illustrate the physical concept behind these three boundary-value problems. More explanatorily, in the Dirichlet Problem, the temperatures are given on the boundary of a solid.

In the Neumann Problem the loss or gain of heat through the boundary is given (it is proportional to $\partial u/\partial n$). In fact in this problem, for a steady-state distribution of temperature, the net flow of thermal energy passing through the boundary of a solid is necessary to be equal to zero, namely,

$$\int_C \psi dC = 0 \ . \qquad \textit{(IV.A.3.1:4)}$$

On the other hand, the Mixed or Robbin's Problem is concerned with the heat exchange with the surrounding medium the temperature of which is ψ/h, where h is the coefficient of thermal conductivity divided by the specific heat (see Mikhlin [*MIKH67*], p.68).

Finally, the remaining category concerns:
(iv) - The Fourth Boundary-Value Problem (Periodic Boundary Problem), which differs from the others in that the solution u has to satisfy some *periodicity* conditions; for example,

$$u\big|_x = u\big|_{x+\ell} \ , \ \frac{\partial u}{\partial n}\Big|_x = \frac{\partial u}{\partial n}\Big|_{x+\ell} \ , \qquad \textit{(IV.A.3.1:5)}$$

where ℓ is called the *period* (see Abdullah [*ABDU83*], p.16).

IV.A.3.2: Mathematical Physics And Well-Posedness Of Problems

The partial differential equations that one is interested in are above all mathematical models of physical phenomena. In fact, for any problem describing a stable situation, one would expect that small variations in the data should result in correspondingly small variations

in the solution. In the case that this did not turn out to be true, then any inclination to accept that the mathematical model of the physical problem has been badly formulated would be entirely justified.

The proposition of finding, in mathematical terms, which problems are, or are not, acceptable models of the physical world has led to the concept of a *well-posed* (or, otherwise, *properly-posed*) problem (see Hadamard [*HADA32*]). This concept is a very important one since problems which are not well-posed cannot, in general, be tackled successfully with numerical methods; however, improperly-posed problems receive also an increasing attention (see Ames [*AMES77*], p.41).

In a definition-like manner, a p.d.e., or a system of p.d.e.'s, is *well-posed in the sense of Hadamard*, if and only if its solution *exists*, is *unique*, and *depends continuously* on the prescribed data. These criteria are physically reasonable in most cases. Existence and uniqueness are an affirmation of the *principle of determinism* without which experiments could not be repeated with the expectation of consistent data, while the continuous dependence criterion is an expression of the stability of the solution.

More analytically, and in brief, this definition, from the aspect of an *initial-value problem*, implies that such a problem is considered well-posed (see Meis and Marcowitz [*MEIS81*], p.2) if it satisfies the conditions:

i) The set of initial values, for which the problem has a solution, is *dense* in the set of all initial values - (*existence*);

ii) for each initial value there exists at most one solution - (*uniqueness*); and,

iii) the solution satisfies a *Lipschitz condition* with respect to

initial values for which the problem is solvable - (*continuity* - in fact, this attribute itself implies uniqueness).

From the aspect of a *boundary-value problem*, the definition implies that it is a well-posed one if it satisfies the following:

i) There exists one and only one solution to the problem which satisfies the boundary conditions - (*existence* and *uniqueness*); and,

ii) small changes in the given functions, which occur in the boundary conditions, cause only small changes in the solution (*continuity* - namely, a continuous dependence of the solutions on the boundary data).

This last requirement is, in particular, necessary if the theoretical results obtained by solving the boundary-value problem are to be utilized in practical applications, where the boundary conditions are known only with whatever degree of accuracy that may be provided by the measuring devices involved. In the case of a well-posed problem, admissible errors in the determination of the boundary conditions do not invalidate the results found; they lead only to insignificant quantitative deviations in the theoretical solution from the experimental results.

To demonstrate the role of the above well-posedness conditions through an example, let us consider the Hadamard's example which is a problem that is *not* well-posed. It concerns the finding of a solution of the two-dimensional Laplace's equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \qquad \textit{(IV.A.3.2:1)}$$

in the semi-strip $y>0$, $-\pi/2\leq x<\pi/2$ under the conditions

$$\left.\begin{aligned} &u\big|_{x=-\pi/2} = u\big|_{x=\pi/2} = 0 \\ &u\big|_{y=0} = 0 \\ &\left.\frac{\partial u}{\partial y}\right|_{y=0} = \phi(x) \qquad (\phi(-\tfrac{\pi}{2}) = \phi(\tfrac{\pi}{2}) = 0) \end{aligned}\right\} . \qquad \textit{(IV.A.3.2:2)}$$

If we set $\phi(x)=0$, the only solution of this problem is $u(x,y)\equiv 0$. On the other hand, if we set

$$\phi(x) = e^{-\sqrt{2n+1}}\cos(2n+1)x \qquad \textit{(IV.A.3.2:3)}$$

the unique solution will be

$$u = \frac{1}{2n+1}\, e^{-\sqrt{2n+1}}\cos(2n+1)x\sinh(2n+1)y . \qquad \textit{(IV.A.3.2:4)}$$

It is easy to show that the function ϕ and all its derivatives, for sufficiently large n, only differ by an arbitrarily slight amount from zero. Yet, for any non-zero constant y, the function u has the form of a cosine function of arbitrarily large amplitude provided n is large. Consequently, for sufficiently large n, this function differs by an arbitrarily great amount from the zero solution (see Mikhlin [*MIKH67*], p.19).

IV.A.3.3: Analytical/Numerical Approximate Methods Of Solution

Although we have repeatedly mentioned the term 'solutions' of p.d.e.'s, we have said almost nothing so far about their solution methods themselves.

In general, for the solution of problems with arbitrarily shaped regions and general prescribed conditions, an exact solution to a given p.d.e. is not usually possible to be determined. Only in the simplest

cases can a solution be analytical either, in implicit form, or by involving a finite formula.

The *Approximate Methods*, which have been developed to tackle this problem, can be divided into two principal groups, the *Analytical* and the *Numerical Methods*. Despite the fact that p.d.e.'s were 'invented' relatively early, in the first half of the eighteenth century, for nearly two hundred years they were utilized mostly as analytical tools to describe the physical world with which they are intimately related. The primary mathematicians'[†] interest was in analytical solutions and elegant analytical methods were developed throughout the eighteenth and nineteenth centuries.

More specifically, by *Analytical Approximate Methods* we mean analytical procedures for obtaining solutions in an analytical form of functions, which satisfy the given equations *at every point* of the region of integration, as well as, the prescribed conditions; in some sense, they are close to the exact solutions of the problems (e.g., the exact solution is in the form of a certain infinite series, while the approximate solution is the sum of the first few terms). This kind of approximate methods may be classified into *three* broad categories, the *asymptotic*, *weighted residual* and the *iterative methods*; certainly, combinations of these methods may and have been utilized to develop alternate *ad hoc* procedures.

Asymptotic methods have at their foundation a desire to obtain solutions that are approximately valid when a physical parameter (or variable) of the problem is very small or very large (e.g., the *perturbation* procedures).

[†] *Starting with d'Alembert in the eighteenth century.*

The *weighted residual methods*,† probably originating in the calculus of variations, ask for the approximate solution to be close to the exact solution in the sense that the difference between them (*residual*) is minimized (e.g., the *collocation* procedure requires the residual to vanish at a set of points).

Finally, the *iterative methods* are analogous to the Picard method of o.d.e.'s in that repetitive calculation, via some operation F whose form is $u_{n+1}=F(u_n,u_{n-1},\ldots)$, successively improves the approximation. Transformation of the equations to an integral equation leads to iteration.

In general, though, there is a wide class of problems for which the analytical solutions are not readily available. For example, the area of integration S can rarely be found such that the initial and boundary conditions are satisfied. The boundary curves C may not be defined at all and even if their equations are known, the boundary conditions may be difficult to be satisfied. Furthermore, it is almost impossible to find an analytical solution if certain changes are made to the shape of the area of integration S, or to the initial and boundary conditions. All these facts implied that numerical techniques had to be sought.

These *Numerical Approximate Methods* fall into a separate category, since the approximate values of the required solution can be found at various points of a region under consideration in a tabular form, as opposed to the functional forms above. However, finding particular solutions numerically, by some more-or-less approximate set of calculations on numbers, was not recognized as a

†*Often called 'direct methods' of the calculus of variations (see Ames [AMES65]).*

respectable pursuit and consequently these methods were not held in high esteem by the scientific community of the time; the principal reason was that the best computational device in hand, at that time, was a desktop mechanical calculator.

A gradual change became perceivable towards the turn of the twentieth century and many numerical approximate methods for p.d.e.'s were to be developed. In fact, Richardson, to wit, found it necessary to write an introduction to his 1910 paper, 'The approximate arithmetical solution ... of physical problems involving partial differential equations, ...', in which, after having noted that there are many physical problems where analytical methods fail, he explains that,

> 'there is a demand for rapid methods... applicable to unusual equations and irregular bodies. If they can be accurate, so much the better; but *1* percent would suffice for many purposes'.

Descriptions of numerical approximate methods began to appear in the literature with an increased frequency. However, the true revolution came with the post *World War II* development of high-speed digital and analog computing devices which spurred a substantial growth, by orders of magnitude, on the scope of the mathematical science of Numerical Analysis. The methods that were utilized at first were mostly adaptations of those that had been developed in precomputer days (intended for pencil and paper implementation), to whatever new capabilities the emerging electronic computers were offering.

Today, there is a number of numerical approximate methods for solving p.d.e.'s which can be distinguished into two different main

classes, the *finite-difference* and the *finite-element* methods.

The method of *finite-differences* is the most widely utilized and understood method for problems in p.d.e.'s, which is supported by the fact that it is the only one of the methods that stands out as being universally applicable to both linear and non-linear problems. The main subclasses of this method are the methods of *lines* and *nets*. To be more specific, in these methods the derivatives occurring in the p.d.e. are replaced by suitable approximations over some small interval and the solution of the resulting finite-difference approximation is sought at a number of discrete points†.

On the other hand, the most important development in the 1950s and 60s, that qualifies as *modern*, was the *finite-element* method, which is out of the scope of the present Thesis. However, in brief, this method rests on more rigorous foundations than the method of finite-differences, and this fact did not escape the attention of numerous mathematicians who have consequently devoted themselves to furthering the supporting theory. Much of the drudgery of generating discretized equations prior to their solutions can now be delegated to finite-element computer codes, in particular for elliptic equations that describe structural problems of mechanical and civil engineering, and steady-state field problems in many other applied sciences. In fact, codes with a similar intent had been implemented with finite-difference methods, but they were largely unable to adapt the discretization of space to the complex shapes and irregularities found in many real-life problems; with finite elements, complex shapes and irregularities create little complication any more, thus making the

†*These methods are, otherwise, called 'discrete' numerical methods.*

corresponding codes convenient to utilize and broad in their applicability.

To conclude, in *Section B*, we confine ourselves to finite-difference methods only as applied to solve parabolic p.d.e.'s; the *Section* continues by presenting a new class of powerful solution methods, i.e. the Group Explicit methods, and by exploiting, in a comparative manner, the potential parallelism of the Standard Explicit method and the new methods, the experimental vehicle being a non-linear parabolic p.d.e. of second-order.

SECTION B

PARALLEL EXPLOITATION OF EXPLICIT METHODS FOR THE SOLUTION OF NON-LINEAR PARABOLIC EQUATIONS

IV.B.1: PRELIMINARY CONCEPTS AND NOTATIONS OF FINITE-DIFFERENCE APPROXIMATIONS TO DERIVATIVES

The ultimate goal of *discrete* numerical methods is the reduction of continuous systems to *equivalent* discrete systems which are suitable for high-speed computer solution. However, one can be, initially, deceived by the, seemingly, elementary nature of these techniques. To be more specific, since their usage is certainly widespread, a little knowledge of them can easily lead to a misapplication; in fact, these approximations raise many serious and difficult mathematical questions of convergence, stability, and consistency.

Discretization of the governing equations and boundary conditions of the continuous problem may be accomplished *physically*, but is more often carried out *mathematically*.

The *physical* approach is sometimes considered by the experts as quite useful, since it can motivate further analysis. In such a *modus operandi* approach the physical characteristics of the continuous system are given to the discrete physical model; for example, a heat-conducting slab could be replaced by a network of heat-conducting rods. A direct

application of the physical laws to the discrete system then gives rise to the governing equations.

On the other hand, in the *mathematical* approach, which we shall follow, the continuous formulation is transformed to one (or more) discrete formulation(s) by replacing the derivatives by finite-difference approximations; in the case that the formulation of the continuous problem is already available, this procedure is simpler and more flexible. Derivatives can be approximated by finite-differences in various ways, but what is a fact is that all such approximations introduce errors, the so-called *truncation errors*[†]. Their presence will be henceforth signified by employing the asymptotic O(big *oh*) notation[‡], introduced by Bachmann and Landau (see Ames [*AMES69*]), which clearly suppresses much less information than the 'limit' notation, being, in addition, easy to handle; in fact this O-notation denotes that, if S is any set and f, ϕ are real or complex functions defined on S, then the notation

$$f(t) = O[\phi(t)], \quad t \in S, \tag{IV.B.1.1}$$

means[*] that a positive number M exists, independent of t, such that

$$|f(t)| \leq M|\phi(t)|, \text{ for all } t \in S. \tag{IV.B.1:2}$$

Thus, the O symbol means 'something that is, in absolute value, at most

---

[†]*Some texts confuse this term with the 'discretization error', which refers to the error in the solution due to the replacement of the continuous p.d.e. by a discrete model; in this Chapter, though, the term 'truncation error' is reserved only for the difference between the differential equation and its approximating difference equation.*

[‡]*This notation was first introduced in (par.-II.B.1) as simply '(term of) order', e.g. h^2, and meant that, 'when h is small enough the term behaves essentially like a constant times h^2'. Here, we make this concept mathematically precise.*

[*]*In certain cases though, to eliminate some non-essential minor inconvenience, a modified O-notation is introduced (see Ames [AMES69], p.12).*

a constant multiple of the absolute value of'; if $\phi(t)\neq 0$, for all $t \in S$, then equation *(IV.B.1:2)* implies that $f(t)/\phi(t)$ is bounded in S.

Returning to finite-differences, let us present the concept behind the so-called *finite-difference grid*. Assume that the solution region of function u, on the x-t plane, is an *open rectangle* $[0,1]\times[0,+\infty)$, which is covered by a rectangular grid (sometimes called a *mesh* or *net*), with grid spacings, $\Delta x=h$, $\Delta t=k$, in the x,t directions, respectively. The values of h,k are assumed uniform throughout the region, albeit they are not necessarily so. The intersection points with coordinates (x_i,t_j)[†] are called *grid points (mesh* or *lattice points*, *nodes* or *pivots)* and the spacings are called *grid lengths* (or, *mesh sizes*).

The grid points (x,t) are given by,

$$x = x_i = i\Delta x = ih,\ i=0,1,2,\ldots,m, \qquad (IV.B.1:3)$$

where $m=1/h$ and,

$$t = t_j = j\Delta t = jk,\ j=0,1,2,\ldots\ ; \qquad (IV.B.1:4)$$

they lie on lines parallel to both axes as shown in *Figure (IV.B.1-f1)* below.

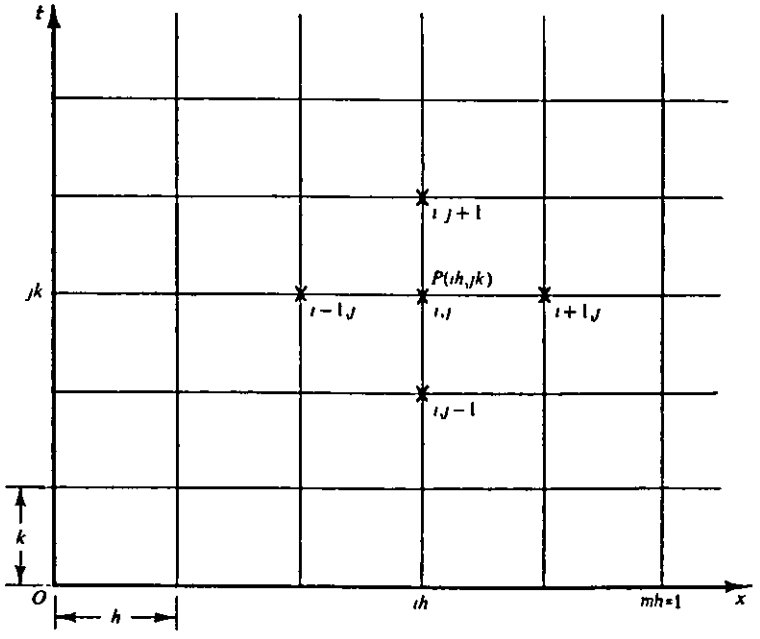


Figure IV.B.1-f1: The Finite-Difference Grid.

[†]*Denoted by* (i,j).

The aim through this finite-difference grid is to seek approximate values of the desired solution of function u at these intersection points; consequently, the problem reduces to the solution of algebraic equations (linear, if the differential equation is linear), which can be obtained in one of several ways.

Finally, the following notations will be utilized for values of u and its derivatives at the grid point $P(ih,jk)$,

$$u_{i,j} = u(x,t) = u(ih,jk) \qquad (IV.B.1:5)$$

$$\left.\begin{aligned} \left.\frac{\partial^r u}{\partial t^r}\right|_{i,j} &= \left.\frac{\partial^r u}{\partial t^r}\right|_{\substack{x=ih \\ t=jk}}, \quad r=1,2,\ldots \\ \left.\frac{\partial^s u}{\partial x^s}\right|_{i,j} &= \left.\frac{\partial^s u}{\partial x^s}\right|_{\substack{x=ih \\ t=jk}}, \quad s=1,2,\ldots \end{aligned}\right\} . \qquad (IV.B.1:6)$$

We shall conclude this paragraph by concentrating on the Taylor's series expansion method, which will be utilized throughout the *Chapter* and is probably the best known of all methods for deriving finite-difference approximations.

Let us assume that $u(x)$ has up to $(k+1)$-order continuous derivatives on $a \leq x \leq b$, for some $k \geq 0$; also that $\xi \in [a,b]$ and h is a real number $\neq 0$ such that $\xi+h \in [a,b]$. Then, Taylor's formula is given by

$$(T_{F_1}): \; u(\xi+h) = u(\xi) + \frac{h}{1!}\frac{du(\xi)}{d\xi} + \frac{h^2}{2!}\frac{d^2u(\xi)}{d\xi^2} + \ldots + \frac{h^k}{k!}\frac{d^ku(\xi)}{d\xi^k} + R_{k+1}, \qquad (IV.B.1:7)$$

where the remainder R_{k+1} is expressed using the derivative of order $(k+1)$ on a point between $\xi, \xi+h$; this remainder can be estimated mainly in the following three different ways:

(*Lagrange*): $$R_{k+1}^{L} = \frac{1}{k!}\int_{\xi}^{\xi+h}(\xi+h-z)^{k}\frac{d^{k+1}u}{dz^{k+1}}\,dz = \frac{h^{k+1}}{(k+1)!}u^{(k+1)}(\xi+\theta h),$$

for some $0\leq\theta\leq 1$; *(IV.B.1:8)*

(*Cauchy*): $$R_{k+1}^{C} = \frac{h^{k+1}}{k!}(1-\delta)^{k}u^{(k+1)}(\xi+\delta h),$$

for some $0\leq\delta\leq 1$; *(IV.B.1:9)*

(*Schlömilch*): $$R_{k+1}^{S} = \frac{h^{k+1}}{k!\rho}(1-\lambda)^{k+1-\rho}u^{(k+1)}(\xi+\lambda h),$$

for some $0\leq\lambda\leq 1$, and ρ integer.

(IV.B.1:10)

If we neglect the remainder in formula (T_{F_1}), then the second part of this formula expresses an approximation of $u(\xi+h)$ under a polynomial form of degree k referring to h, namely,

$$u(\xi+h) \cong^{\dagger} T_{F_1}^{k}(h) = u(\xi) + \frac{h}{1!}\frac{du(\xi)}{d\xi} + \frac{h^2}{2!}\frac{d^2u(\xi)}{d\xi^2} + \ldots + \frac{h^k}{k!}\frac{d^ku(\xi)}{d\xi^k}.$$

(IV.B.1:11)

In the case when there are two (or more) independent variables, then the derivatives become partial derivatives and the formula (T_{F_1}) is slightly modified as follows: Let (x_0,t_0) and (x_0+h,t_0+k) be given points and assume that $u(x,t)$ is $(k+1)$ times continuously differentiable for all (x,t) in some neighbourhood of $L(x_0,t_0;x_0+h,t_0+k)$[‡]. Then,

$$(T_{F_2}): u(x_0+h,t_0+k) = u(x_0,t_0) + \sum_{j=1}^{k}\frac{1}{j!}\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial t}\right)^{j}u(x,t)\Big|_{\substack{x=x_0\\t=t_0}} + R_{k+1},$$

(IV.B.1:12)

where the remainder term is similarly, as before, expressed by the

---

[†] *We use '≅' as the symbol representing approximation.*

[‡] *It denotes the set of all points (x,t) on the straight line segment joining (x_0,t_0) and (x_0+h,t_0+k).*

partial derivatives of order $(k+1)$, i.e.,

$$R_{k+1} = \frac{1}{(k+1)!}\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial t}\right)^{k+1} u(x,t)\Big|_{\substack{x=x_0+\xi h,\\ t=t_0+\xi k}} \text{ for some } 0\leq\xi\leq 1, \qquad (IV.B.1:13)$$

and the point $(x_0+\xi h, t_0+\xi k)$ is an unknown point of the line $L(x_0,t_0; x_0+h, t_0+k)$; in other terms that is,

$$R_{k+1} = O[(|h|+|k|)^{k+1}] \ , \qquad (IV.B.1:14)$$

which, according to what was discussed previously in the present paragraph, means that there exists a positive number M such that,

$$|R_{k+1}| \leq M(|h|+|k|)^{k+1} \ , \qquad (IV.B.1:15)$$

as both h,k tend to zero.

IV.B.1.1: Parabolic Equations In One Space Dimension And Discretizing Finite-Difference Formulae

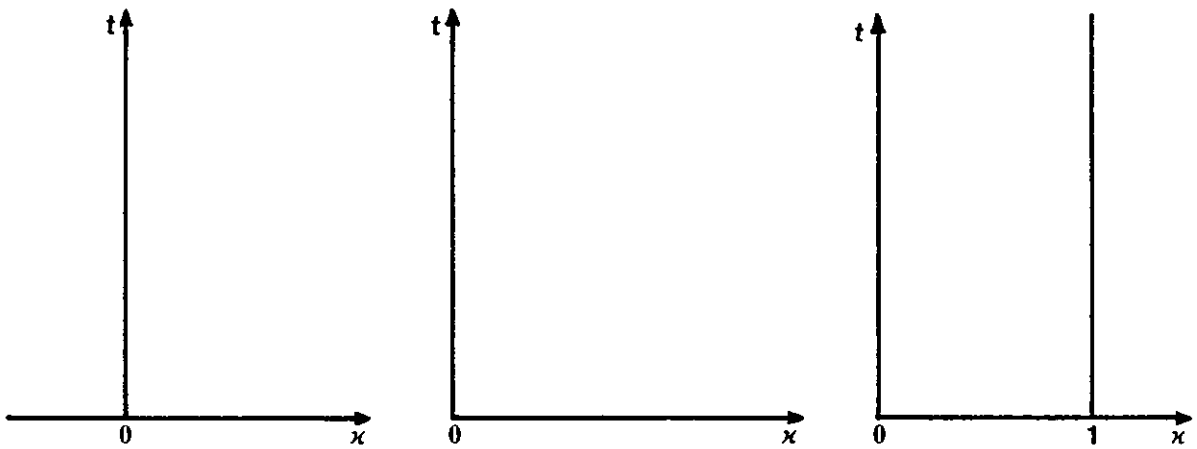
Prior to proceeding with the formulation of the finite-difference approximations for parabolic p.d.e.'s, let us discuss first, briefly, the usual region of solution for such equations, a characteristic representative of which is given by the heat-conduction equation *(IV.A.3:11)*.

In general, a plethora of problems in Physics and Engineering science, requiring numerical solution, involve special cases of the linear parabolic p.d.e.

$$\sigma(x,t)\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(a(x,t)\frac{\partial u}{\partial x}\right) + b(x,t)\frac{\partial u}{\partial x} - c(x,t)u \ , \qquad (IV.B.1.1:1)$$

which holds within some prescribed region Ω on the x-t plane; within this region, the functions σ, a are strictly positive and c is non-

negative. The region of solution is usually one of the three forms illustrated in *Figure (IV.B.1.1-f1)* below.



(i): $(x,t)\in(-\infty,+\infty)\times[0,+\infty)$ *(ii):* $(x,t)\in[0,+\infty)\times[0,+\infty)$ *(iii):* $(x,t)\in[0,1]\times[0,+\infty)$

Figure IV.B.1.1-f1: The Solution Regions for Parabolic Equations.

The case *(i)* is called the *semi-infinite plane*. This leads to a purely initial-value (Cauchy) problem consisting of equation *(IV.B.1.1:1)* and the initial condition

$$u = f(x), \text{ for } -\infty<x<+\infty \text{ at } t=0. \qquad \textit{(IV.B.1.1:2)}$$

The case *(ii)* is called the *quarter plane*. This leads to an initial/boundary-value problem consisting of equation *(IV.B.1.1:1)* together with the initial condition

$$u = f(x), \text{ for } 0\leq x<+\infty \text{ at } t=0\ , \qquad \textit{(IV.B.1.1:3)}$$

and the boundary condition[†]

$$\alpha_0(x,t)u+\alpha_1(x,t)\frac{\partial u}{\partial x} = \alpha_2(x,t), \text{ at } x=0,\ t\geq 0\ , \qquad \textit{(IV.B.1.1:4)}$$

where $$\alpha_0(0,t)\geq 0,\ \alpha_1(0,t)\leq 0, \text{ and } \alpha_0-\alpha_1>0\ . \qquad \textit{(IV.B.1.1:5)}$$

[†]*Some boundary information is also required at $x=\infty$, $t\geq 0$.*

Finally, the case *(iii)* is called the *open rectangle plane*. This again leads to an initial/boundary-value problem consisting of equation *(IV.B.1.1:1)* together with the initial condition

$$u = f(x), \text{ for } 0 \leq x \leq 1 \text{ at } t=0 , \qquad \textit{(IV.B.1.1:6)}$$

and the boundary conditions

$$\left.\begin{aligned} &\alpha_0(x,t)u+\alpha_1(x,t)\frac{\partial u}{\partial x} = \alpha_2(x,t) \text{ at } x=0,\ t\geq 0 \\ &\beta_0(x,t)u+\beta_1(x,t)\frac{\partial u}{\partial x} = \beta_2(x,t) \text{ at } x=1,\ t\geq 0 \\ &\text{where} \quad \beta_0(1,t)\geq 0,\ \beta_1(1,t)\geq 0,\ \beta_0-\beta_1>0 \text{ and} \end{aligned}\right\} \qquad \textit{(IV.B.1.1:7)}$$

similar conditions on the α's as in *(IV.B.1.1: 5)*.

Returning to finite-difference approximations to derivatives, let us assume that a function u and its derivatives are single-valued, finite and continuous functions of x; then, using Taylor's formula *(IV.B.1:7)*, we obtain,

$$u(x+h) = u(x)+h\frac{du}{dx} + \frac{h^2}{2}\frac{d^2u}{dx^2} + \frac{h^3}{6}\frac{d^3u}{dx^3} + \cdots \qquad \textit{(IV.B.1.1:8)}$$

and similarly,

$$u(x-h) = u(x)-h\frac{du}{dx} + \frac{h^2}{2}\frac{d^2u}{dx^2} - \frac{h^3}{6}\frac{d^3u}{dx^3} \cdots . \qquad \textit{(IV.B.1.1:9)}$$

Addition of these expansions results in

$$u(x+h)+u(x-h) = 2u(x)+h^2\frac{d^2u}{dx^2} + O(h^4) , \qquad \textit{(IV.B.1.1:10)}$$

where the quantity $O(h^4)$ represents the asymptotic notation for the *truncation error* of this approximation and denotes terms containing fourth[†] and higher powers of h. Assuming that these are negligible compared to lower powers of h results in

[†]*Leading term.*

$$\left.\frac{d^2u}{dx^2}\right|_{x=x} \cong \frac{u(x+h)-2u(x)+u(x-h)}{h^2} \qquad (IV.B.1.1:11)$$

with a truncation error of order h^2 - (second-order approximation)[†].

In a similar way, subtracting equation *(IV.B.1.1:9)* from equation *(IV.B.1.1:8)* and neglecting terms of order h^3 results in

$$\left.\frac{du}{dx}\right|_{x=x} \cong \frac{u(x+h)-u(x-h)}{2h} , \qquad (IV.B.1.1:12)$$

with a truncation error of order h^2.

As shown in *Figure (IV.B.1.1-f2)*, equation *(IV.B.1.1:12)* clearly approximates the slope of the tangent at P by the slope of the chord AB, and is called a *central-difference* approximation. The slope of the tangent at P can also be approximated by either the slope of the chord PB, resulting in the *forward-difference* formula

$$\left.\frac{du}{dx}\right|_{x=x} \cong \frac{u(x+h)-u(x)}{h} , \qquad (IV.B.1.1:13)$$

or the slope of the chord AP, resulting in the *backward-difference* formula

$$\left.\frac{du}{dx}\right|_{x=x} \cong \frac{u(x)-u(x-h)}{h} . \qquad (IV.B.1.1:14)$$

Both *(IV.B.1.1:13,14)* can be written directly from equations *(IV.B.1.1:8,9)*, respectively, assuming second and higher powers of h are negligible; this shows that the truncation errors in these formulae are both $O(h)$.

In the case of two (or more) independent variables the above formulae are slightly modified[‡]. For example, let us consider a representative grid point (i,j); then by applying formula *(IV.B.1:12)* we obtain,

---

[†] *If $U(x)$ is an approximation to $u(x)$, we say it is of order 'n', with respect to some quantity $\Delta x=h$, if 'n' is the largest possible positive real number such that $|u-U|=O(h^n)$, as $h\to 0$.*

[‡] *In fact, the new formulae can be correspondingly derived from the previous formulae in a straight-forward manner.*

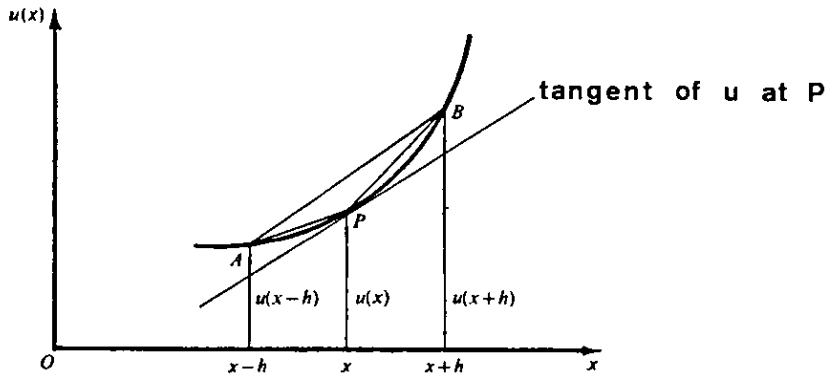


Figure IV.B.1.1-f2: Geometrical Representation of the Finite-Difference Formulae Concept.

$$u_{i\pm1,j} = u_{i,j} \pm h\left.\frac{\partial u}{\partial x}\right|_{i,j} + \frac{h^2}{2!}\left.\frac{\partial^2 u}{\partial x^2}\right|_{i\pm\xi_1,j} \quad , \; 0<\xi_1<1 \qquad (IV.B.1.1:15)$$

and

$$u_{i,j\pm1} = u_{i,j} \pm k\left.\frac{\partial u}{\partial t}\right|_{i,j} + \frac{k^2}{2!}\left.\frac{\partial^2 u}{\partial t^2}\right|_{i,j\pm\xi_2} \quad , \; 0<\xi_2<1, \qquad (IV.B.1.1:16)$$

where expansions about $u_{i,j}$ up to the second power of h,k have been considered. Consequently, for the first-order spatial derivative we obtain the following formulae,

(*forward-difference*):
$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{u_{i+1,j}-u_{i,j}}{h} + O(h) \quad , \qquad (IV.B.1.1:17)$$

(*backward-difference*):
$$\left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{u_{i,j}-u_{i-1,j}}{h} + O(h) \quad . \qquad (IV.B.1.1:18)$$

If $u_{i+1,j}, u_{i-1,j}$ are expanded about $u_{i,j}$ up to the third power of h, i.e.,

$$u_{i\pm1,j} = u_{i,j} \pm h\left.\frac{\partial u}{\partial x}\right|_{i,j} + \frac{h^2}{2!}\left.\frac{\partial^2 u}{\partial x^2}\right|_{i,j} \pm \frac{h^3}{3!}\left.\frac{\partial^3 u}{\partial x^3}\right|_{i\pm\xi_1,j} \quad , \; 0<\xi_1<1,$$

(IV.B.1.1:19)

then by subtracting them the following formula is obtained,

$$(\textit{central-difference}): \quad \left.\frac{\partial u}{\partial x}\right|_{i,j} = \frac{u_{i+1,j} - u_{i-1,j}}{2h} + O(h^2) \ . \qquad (IV.B.1.1:20)$$

Finite-difference formulae for the higher-order derivatives can also be developed in a similar way; for example, from

$$u_{i\pm1,j} = u_{i,j} \pm h\left.\frac{\partial u}{\partial x}\right|_{i,j} + \frac{h^2}{2!}\left.\frac{\partial^2 u}{\partial x^2}\right|_{i,j} \pm \frac{h^3}{3!}\left.\frac{\partial^3 u}{\partial x^3}\right|_{i,j} + \frac{h^4}{4!}\left.\frac{\partial^4 u}{\partial x^4}\right|_{i\pm\xi_1,j},$$

$$0<\xi_1<1, \qquad (IV.B.1.1:21)$$

we obtain the formula,

$$(\textit{central-difference}): \quad \left.\frac{\partial^2 u}{\partial x^2}\right|_{i,j} = \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2} + O(h^2). \qquad (IV.B.1.1:22)$$

In a similar manner the finite-difference formulae for the derivatives of u with respect to the independent variable t at the grid point (i,j) may be obtained. For example, the *forward-*, *backward-* and *central-difference* formulae for the first-order time derivative are, respectively, given by,

$$\left.\frac{\partial u}{\partial t}\right|_{i,j} = \frac{u_{i,j+1} - u_{i,j}}{k} + O(k) \qquad (IV.B.1.1:23)$$

$$\left.\frac{\partial u}{\partial t}\right|_{i,j} = \frac{u_{i,j} - u_{i,j-1}}{k} + O(k) \qquad (IV.B.1.1:24)$$

$$\left.\frac{\partial u}{\partial t}\right|_{i,j} = \frac{u_{i,j+1} - u_{i,j-1}}{2k} + O(k^2) \ ; \qquad (IV.B.1.1:25)$$

while the second-order derivative, correspondingly to formula *(IV.B.1.1:22)*, is given by the *central-difference* formula,

$$\left.\frac{\partial^2 u}{\partial t^2}\right|_{i,j} = \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{k^2} + O(k^2) \ . \qquad (IV.B.1.1:26)$$

With the derivatives being discretized with the above finite-difference approximations, it is instructive, at this stage, to establish a criterion for the *local accuracy* of a finite-difference equation. In particular, let us consider a special case of equation *(IV.B.1.1:1)*, i.e. the heat-conduction equation

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \text{ (with constant coefficients).} \qquad \textit{(IV.B.1.1:27)}$$

By the use of equations *(IV.B.1.1:22,23)* we can, respectively, substitute the *right*-and *left-hand-side* terms of this equation; hence, we obtain,

$$\frac{u_{i,j+1}-u_{i,j}}{k} = \frac{u_{i+1,j}-2u_{i,j}+u_{i-1,j}}{h^2} + O(k+h^2), \qquad \textit{(IV.B.1.1:28)}$$

which can be approximately and explicitly written as

$$U_{i,j+1} = (1-2r)U_{i,j} + r(U_{i+1,j}+U_{i-1,j}) , \qquad \textit{(IV.B.1.1:29)}$$

where $r=k/h^2$[†] and $U_{i,j}$ is an approximation to $u_{i,j}$.

Such approximation methods are called *explicit(open)*, because their formulae involve only *one* grid point at the advanced time-level $t=(j+1)k$; whereas formulae involving more than one point at this advanced time-level introduce the so-called *implicit (closed)* methods. Since parabolic (and hyperbolic) equations characteristically have open integration domains, explicit methods are applicable to these problems; stability questions are critical in these situations though, while stability difficulties are not as serious in implicit methods.

Now, without suppressing any information via the asymptotic notation, the substitution of formulae *(IV.B.1.1:22,23)* to equation *(IV.B.1.1:27)* results in

[†]*This is called the 'grid ratio'.*

$$u_{i,j+1} - (1-2r)u_{i,j} - r(u_{i+1,j} + u_{i-1,j}) =$$

$$k\left(\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2}\right)_{i,j} + \tfrac{1}{2}k^2\left(\frac{\partial^2 u}{\partial t^2} - \frac{1}{6r}\frac{\partial^4 u}{\partial x^4}\right)_{i,j} + \ldots \qquad (IV.B.1.1:30)$$

Let us now introduce the difference between the exact solutions of the differential and difference equations at the grid point (i,j) as:

$$e_{i,j} = u_{i,j} - U_{i,j} \quad (\textit{local discretization error}); \qquad (IV.B.1.1:31)$$

then from equations *(IV.B.1.1:27,29,30)* the result

$$e_{i,j+1} = (1-2r)e_{i,j} + r(e_{i+1,j} + e_{i-1,j}) + \tfrac{1}{2}k^2\left(\frac{\partial^2 u}{\partial t^2} - \frac{1}{6r}\frac{\partial^4 u}{\partial x^4}\right)_{i,j} + \ldots$$

(IV.B.1.1.32)

is obtained. The quantity

$$\tfrac{1}{2}k^2\left(\frac{\partial^2 u}{\partial t^2} - \frac{1}{6r}\frac{\partial^4 u}{\partial x^4}\right)_{i,j} + \ldots \qquad (IV.B.1.1.33)$$

is defined as the *local truncation error*[†][‡] (henceforth abbreviated as *l.t.e.*) of equation *(IV.B.1.1:29)* and the *principal part* of the l.t.e. is

$$\tfrac{1}{2}k^2\left(\frac{\partial^2 u}{\partial t^2} - \frac{1}{6r}\frac{\partial^4 u}{\partial x^4}\right)_{i,j} . \qquad (IV.B.1.1:34)$$

To conclude, the concept of l.t.e. is absolutely necessary for the discussion of the concepts included in the following paragraph.

IV.B.1.2: A Descriptive Treatment Of The Convergence, Stability, And Consistency Or Compatibility Concepts

This paragraph is concerned with the conditions that must be

[†] *The 'local order of accuracy' (often abbreviated to local accuracy) of equation (IV.B.1.1:29), as shown, is $O(k+h^2)$. This should not be confused with the 'global accuracy' of a difference equation, which is a measure of the accuracy of the difference equation 'all over' the region under consideration and is a very difficult quantity to estimate.*

[‡] *This is the amount by which the exact solution u of the p.d.e. does not satisfy the difference equation at the grid point (i,j).*

satisfied in order to obtain a reasonably accurate approximation to a p.d.e.'s solution[†] by the solution of the corresponding finite-difference equations. More specifically, these conditions are primarily associated with two different but interrelated problems. The first problem concerns the *convergence* of the exact solution of the approximating difference equations to the exact solution of the differential equation, as h and k tend to zero; while, the second problem concerns the *stability* of the difference scheme resulting from the unbounded growth or controlled decay of any errors associated with its solution.

A difference scheme is said to be *convergent* if the discretization error tends to zero, as $h\to 0$, $k\to 0$. The magnitude of this error, at any grid point, depends on the finite-sizes of the grid lengths h,k, as well as on the number of terms in the truncated series of differences which approximate the derivatives. The order of the overall discretization error is the order of the leading term in the truncated series of differences. The discretization error can be analyzed in terms of the preceding l.t.e.'s; more specifically, as can be seen in the previous paragraph, the l.t.e. at the grid point (i,j) is a measure of the (local) discretization error at the grid point $(i,j+1)$, when the finite-difference scheme is applied once only to the exact solution values of the differential equation, all arithmetic being exact, i.e. without *round-off errors*[‡] The discretization error can usually be diminished by decreasing h,k subject invariably to some relationship between them; but, as this leads to an increase in the number of equations to be solved, this method of improvement is limited by factors such as,

---

[†]*In our case a parabolic p.d.e.'s solution.*

[‡]*They result from the fact that computers carry numbers with a finite number of digits.*

computational cost, computer storage requirements, etc.

In a more 'analytical' treatment of *convergence*, let E_j denote the maximum value of $|e_{i,j}|$ along the j^{th} time-level. When $r \leq \frac{1}{2}$, then equation *(IV.B.1.1:32)*, since all the coefficients of e are positive or zero, gives,

$$\begin{aligned} |e_{i,j+1}| &\leq (1-2r)|e_{i,j}| + r|e_{i+1,j}| + r|e_{i-1,j}| + M[k^2+kh^2] \\ &\leq (1-2r)E_j + rE_j + rE_j + M[k^2+kh^2] \\ &= E_j + M[k^2+kh^2] \ , \end{aligned}$$

where M is the maximum modulus of the expression which is included in the $O[k^2+kh^2]$ for all i,j. As the above is true for all values of i, it is true for $max|e_{i,j+1}|$. Hence,

$$E_{j+1} \leq E_j + M[k^2+kh^2] \leq (E_{j-1} + M[k^2+kh^2]) + M[k^2+kh^2] = E_{j-1} + 2M[k^2+kh^2] \ ,$$

etc., from which it follows that,

$$E_j \leq E_0 + jM[k^2+kh^2] = jM[k^2+kh^2] \ , \qquad \textit{(IV.B.1.2:1)}$$

because the exact solutions u,U, of the differential and difference equations respectively, have the same magnitude† on the initial line, i.e.,

$$E_0 = \max_i |e_{i,0}| = 0 \ .$$

Thus, it follows that,

$$\lim_{\substack{h\to 0 \\ k\to 0}} E_j \leq \lim_{\substack{h\to 0 \\ k\to 0}} jM[k^2+kh^2] = 0 \ . \qquad \textit{(IV.B.1.2:2)}$$

Consequently, under the condition $r \leq \frac{1}{2}$ this error term tends to zero, as $h \to 0$ and $k \to 0$, and the exact solution of the difference equation *(IV.B.1.1:29)* converges to the exact solution of the p.d.e. *(IV.B.1.1:27)*; however, the necessary condition $r \leq \frac{1}{2}$ is a rather severe condition for convergence.

---

†*Their initial values are the same.*

In general, the problem of convergence is a difficult one to investigate efficiently because the final expression for the discretization error is usually in terms of unknown derivatives for which no bounds can be estimated. However, to our benefit, the convergence of difference equations approximating linear parabolic (and hyperbolic) equations can be investigated in terms of stability and consistency or compatibility, which are much easier to deal with.

As it was remarked earlier, in practice, during the solution of the finite-difference equations, which are the equations that are actually solved, each calculation, depending on the particular computer, is carried out to a finite number of decimal places or significant figures, and so an inevitable *round-off error* is introduced at every computational step. Consequently, instead of finding the exact solution of a difference equation, which would be normally found if it were possible to carry out all calculations to an infinite number of decimal places, the actually computed solution is not U, but, say, U^*; this solution will be called the *numerical solution* of the difference equation, while the difference $R=U-U^*$ the *global round-off error*. The *total error* $\Lambda_{i,j}$ at the grid point (i,j) is

$$\Lambda_{i,j} = u_{i,j} - U^*_{i,j} = [(u_{i,j} - U_{i,j}) + (U_{i,j} - U^*_{i,j})]^{\dagger} . \qquad (IV.B.1.2:3)$$

Under the assumption that the discretization error can be controlled, it would seem reasonable, if the growth of the global round-off error $R_{i,j}$ is bounded, for all i, as j tends to infinity, to claim that the difference equations are *stable*; this is not

---

†*The discretization error + the global round-off error.*

applicable though, since the global round-off error depends not only on the difference equations themselves, but also on the particular manner in which different computers round-off numbers and carry out arithmetic.

This error, invariably smaller than the discretization error due to the powerful modern computers, differs from it in that as the grid lengths tend to zero the number of arithmetic operations increases, a fact which implies that this error cannot be made to converge to zero.

Stability can be successfully defined in terms of the boundedness of the exact solution of the difference equations, since it involves only known coefficients and boundary values, is amenable to mathematical analysis and has some useful consequences. More specifically, when propagating the solution forward in time, if $U_{i,j}$ is bounded as j increases it implies that the magnification of each round-off error is also bounded because the same arithmetic operations are applied to both numbers. Also, because $u_{i,j}$ is a fixed number for a given equation with known boundary and initial conditions, when $U_{i,j}$ is bounded then the discretization error is also bounded. As the latter error can be expressed in terms of l.t.e.'s, the difference equations are sometimes defined as stable when local round-off errors and l.t.e.'s do not increase unboundedly as the calculation time-levels increase. This definition of stability, for linear differential equations, is related to convergence through the concept of consistency; more explanatorily, since in actual computation h,k are kept constant during the propagation of the solution forward in time, the conditions necessary for the boundedness of $U_{i,j}$ (and of any form of local error) as j tends to infinity, together with consistency, guarantee convergence.

There are two standard methods of investigating the boundedness of the solution of the finite-difference equations, the *matrix* and the *Fourier series* methods. In the former method, the equations are expressed in matrix form and the eigenvalues of an associated matrix are examined. In the latter and easier method, since no knowledge of matrix algebra is required, a finite Fourier series is used; however, it is less rigorous than the former method, since it neglects the boundary conditions.

Finally, *consistency* or *compatibility* is concerned with the finding of the condition for which a discrete problem is an approximation of the corresponding continuous problem. In practice, sometimes, it is possible that the solution of a stable finite-difference scheme, which approximates a parabolic (or hyperbolic) equation as the grid lengths tend to zero, to converge to the solution of a different differential equation. Such a difference scheme is said to be inconsistent (or incompatible) with the p.d.e. Lax's equivalence theorem (see Richtmyer and Morton [*RICH67*]) gives us the real importance of the concept of consistency; it states that if a linear finite-difference equation is consistent with a properly-posed linear initial-value problem, then stability is the necessary and sufficient condition for convergence.

Consistency can be defined in either of two equivalent but slightly different ways, the more general one being as follows: Let us consider that $L(u)=0$ and $F(U)=0$ represent the p.d.e. and the approximating finite-difference equation in the independent variables x,t, with exact solutions u,U, respectively. Let also v be a continuous function of x,t with a sufficient number of continuous

derivatives to enable $L(v)$ to be evaluated at the grid point (i,j).

The truncation error $T_{i,j}(v)$ at the grid point (i,j) is defined by

$$T^{\dagger}_{i,j}(v) = F(v_{i,j}) - L(v_{i,j}) \ . \qquad (IV.B.1.2:4)$$

Then, if $T_{i,j}(v) \to 0$ as $h \to 0$, $k \to 0$, the difference equation is said to be *consistent* with the p.d.e.

However, to conclude, most authors put $v=u$ because $L(u)=0$; consequently, from equation *(IV.B.1.2:4)* it follows that,

$$T_{i,j}(u) = F(u_{i,j}) \ , \qquad (IV.B.1.2:5)$$

and the truncation error coincides with the l.t.e.[‡] The consistency then of the difference equation relies on the limiting value of the l.t.e. being zero, as $h \to 0$, $k \to 0$.

[†] *With this definition $T_{i,j}$ gives an indication of the error resulting from the replacement of $L(v_{i,j})$ by $F(v_{i,j})$.*

[‡] *See Smith [SMIT78], p.77.*

IV.B.2: Various Finite-Difference Approximation Schemes To A Non-Linear Parabolic Problem

As we have previously discussed in the Thesis parallelism can arise at many different levels within a given problem, which if properly exposed can be efficiently exploited by the new parallel computers.

To summarize, some of the broadly known techniques of performing this are:

(i) The *Vectorization* of the existing software, usually achieved by altering the order to evaluation of terms in a complicated expression, so that a vector or matrix of elements can be handled in one operation;

(ii) the *Divide-and-Conquer* strategy, where the problem is decomposed into a number of independent sub-problems all of which can proceed independently to yield the answer of the original problem; and finally,

(iii) the so termed *Implicit Parallelism* (e.g. recursive decoupling, cyclic reduction), which involves the discovery of independent sub-expressions in the computation capable of proceeding in parallel.

Some other techniques, such as, *Pipelining*, *Broadcasting*, and *Streaming*, are in fact out of our concern since they are more usually associated with the hardware features of the system.

However, another significant technique to exploit the potential parallelism in a numerical algorithm is by the utilization of *Explicit methods*, which are the oldest methods for the solution of many problems. Unfortunately, they suffer from major defects, such as, poor stability and convergence characteristics and require unacceptable lengthy solution times.

Undoubtedly the newer *Implicit methods* are far better, but often we are not able to exploit to the full the Implicit Parallelism within the algorithm.

The principal aim of this *Chapter* is the formulation of some new Explicit methods of solution and the exploitation, in a comparative manner, of the potential parallelism of the Standard Explicit method and the new methods, which, in addition, exhibit improved stability and convergence characteristics. All methods will be formulated for the solution of the one-dimensional non-linear parabolic p.d.e.

$$\frac{\partial u}{\partial t} = G(x,t,u,\frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}), \text{ for } 0 \leqslant x \leqslant 1, \quad t \geqslant 0, \qquad \textit{(IV.B.2:1)}$$

while, for practical purposes, they will also be formulated for, and analytically experimented on, equation

$$\frac{\partial u}{\partial t} = \varepsilon \frac{\partial^2 u}{\partial x^2} - u \frac{\partial u}{\partial x}, \text{ for } \varepsilon > 0 ; \qquad \textit{(IV.B.2:2)}$$

this particular equation has been discussed by Burgers [*BURG48*] as a mathematical model of turbulence and by Cole, et al [*COLE51*] as the approximate theory for weak non-stationary shock waves in a real fluid. This equation can also be considered as a simplified form of the Navier-Stokes equation (see Ames [*AMES65*]).

For decades equation *(IV.B.2:2)* has attracted the attention of many researchers and as a result many finite-difference and finite-element methods have been proposed to solve this equation. One common difficulty with the existing solutions is that as the value of ε decreases a finer grid of points has to be chosen in order to obtain a reasonable accuracy. This in turn determines the problem as prohibitively time consuming and often unrealistic to solve on standard

sequential computers. However, since the emergence of Supercomputers able to execute at rates between 100-500 million floating point operations, it has been feasible to design effective solution methods for problems which previously were not possible to solve.

In the following we shall briefly investigate the existing schemes for the solution of our problem, with the initial condition,

$$u(x,0) = f(x) \quad , \text{ for } 0 \leqslant x \leqslant 1 \qquad \textit{(IV.B.2:3)}$$

and boundary conditions,

$$\left.\begin{aligned} u(0,t) &= g_1(t) \\ u(1,t) &= g_2(t) \end{aligned}\right\} \text{ for } t>0 \qquad \textit{(IV.B.2:4)}$$

A *Standard Explicit* scheme for solving equation *(IV.B.2:1)* is formed by taking the *central-difference* approximations for $\partial u/\partial x$ and $\partial^2 u/\partial x^2$, and a *forward-difference* approximation for $\partial u/\partial t$. This results in a non-linear finite-difference equation of form,

$$U_{i,j+1} = U_{i,j} + kG\left(ih, jk, U_{i,j}, \frac{U_{i+1,j}-U_{i-1,j}}{2h}, \frac{U_{i+1,j}-2U_{i,j}+U_{i-1,j}}{h^2}\right),$$

where again, *(IV.B.2:5)*

$$\left.\begin{aligned} &\Delta x = h,\ \Delta t = k,\ U_{i,j} = U(ih,jk) \\ &x = ih,\ i=0,1,2,\ldots,m,\ m=1/h \\ &t = jk,\ j=0,1,2,\ldots \end{aligned}\right\} \qquad \textit{(IV.B.2:6)}$$

However, $U_{i,j+1}$ is a non-linear function in known terms $U_{i-1,j}$, $U_{i,j}$ and $U_{i+1,j}$ and therefore it can be computed directly without involving an iteration process.

For the equation *(IV.B.2:2)* this classical explicit scheme is given by the formula,

$$U_{i,j+1} = \varepsilon r(U_{i+1,j}-2U_{i,j}+U_{i-1,j}) + \left[1-\frac{rh}{2}(U_{i+1,j}-U_{i-1,j})\right]U_{i,j},$$

$$\text{for } r=k/h^2 \ . \qquad \textit{(IV.B.2:7)}$$

Although this method is computationally simple it suffers from a very restrictive stability condition, i.e. the value of r should be $\leq \frac{1}{2}$ in order to retain reasonable accuracy; however, in practice, a very small value of ε will force a small value of h, which in turn, with $r \leq \frac{1}{2}$, makes the value of k too small for practical purposes. A clear advantage of this method is the independency of the points within a single time-level, which makes it naturally suitable for *SIMD* type architectures (see Evans, et al [*EVAN83a*]). Finally, the molecular diagram of formula *(IV.B.2:7)*, which is for illustrative purposes rewritten as

$$U_{i,j+1} = r(\varepsilon+\frac{h}{2}U_{i,j})U_{i-1,j} + (1-2\varepsilon r)U_{i,j} + r(\varepsilon-\frac{h}{2}U_{i,j})U_{i+1,j}, \qquad \textit{(IV.B.2:8)}$$

is as in *Figure (IV.B.2-f1)*, where,

$$\left.\begin{aligned} a &= \varepsilon - \frac{h}{2}U_{i,j} \\ b &= \varepsilon + \frac{h}{2}U_{i,j} \end{aligned}\right\}. \qquad \textit{(IV.B.2:9)}$$

A stable *Fully Implicit* scheme can be obtained by taking the *backward-difference* approximation to $\partial u/\partial t$, at point $(i,j+1)$, to obtain,

$$U_{i,j+1} = U_{i,j} + kG\left(ih,(j+1)k,U_{i,j+1},\frac{U_{i+1,j+1}-U_{i-1,j+1}}{2h},\frac{U_{i+1,j+1}-2U_{i,j+1}+U_{i-1,j+1}}{h^2}\right) \qquad \textit{(IV.B.2:10)}$$

in case of equation *(IV.B.2:1)*, and

$$U_{i,j+1} = U_{i,j} + \varepsilon r(U_{i+1,j+1}-2U_{i,j+1}+U_{i-1,j+1}) - \frac{rh}{2}(U_{i+1,j+1}-U_{i-1,j+1})U_{i,j+1} \qquad \textit{(IV.B.2:11)}$$

for the particular case of equation *(IV.B.2:2)*. In this approximation

scheme the unknown $U_{i-1,j+1}$, $U_{i,j+1}$, $U_{i+1,j+1}$ are involved in a non-linear relation and therefore it must be solved iteratively. The molecular diagram of formula *(IV.B.2:11)*, which is for illustrative

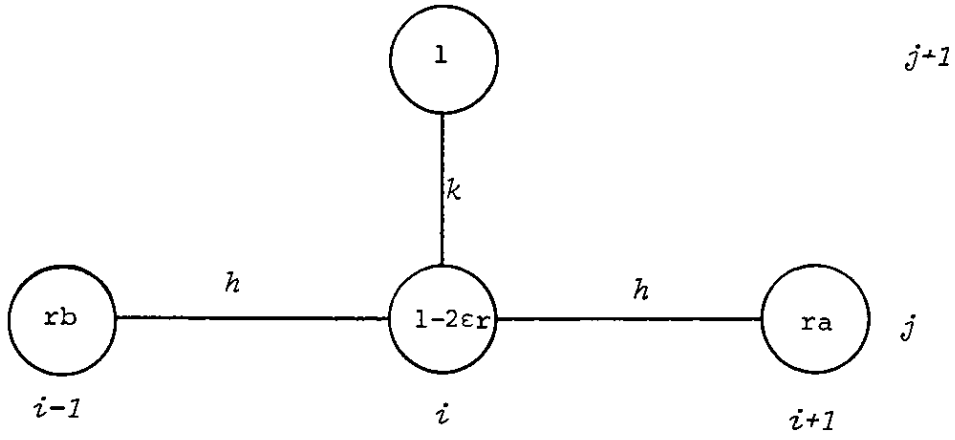


Figure IV.B.2-f1: The Molecular Diagram of the *Standard Explicit* Scheme for the Direct Solution of Burgers' Equation.

purposes rewritten as

$$-r(\varepsilon+\frac{h}{2}U_{i,j+1})U_{i-1,j+1}+(1+2\varepsilon r)U_{i,j+1}-r(\varepsilon-\frac{h}{2}U_{i,j+1})U_{i+1,j+1}=U_{i,j}, \qquad (IV.B.2:12)$$

is as in *Figure (IV.B.2-f2)*, where,

$$\left.\begin{aligned} a &= \varepsilon-\frac{h}{2}U_{i,j+1} \\ b &= \varepsilon+\frac{h}{2}U_{i,j+1} \end{aligned}\right\}. \qquad (IV.B.2:13)$$

Alternatively, approximating the partial derivatives $\partial u/\partial x$ and $\partial^2 u/\partial x^2$ by the *mean* of their *central-difference* approximations on the j^{th} and $(j+1)^{th}$ time-level will result in a *Crank-Nicolson* type of implicit scheme; namely,

$$U_{i,j+1}=U_{i,j}+kG\left[ih,(j+\tfrac{1}{2})k,\frac{U_{i,j}+U_{i,j+1}}{2},\tfrac{1}{2}\frac{(U_{i+1,j}-U_{i-1,j})}{2h}+\tfrac{1}{2}\frac{(U_{i+1,j+1}-U_{i-1,j+1})}{2h},\tfrac{1}{2}\frac{(U_{i+1,j}-2U_{i,j}+U_{i-1,j})}{h^2}+\right.$$

$$+ \tfrac{1}{2} \frac{(U_{i+1,j+1} - 2U_{i,j+1} + U_{i-1,j+1})}{h^2}] \quad (IV.B.2:14)$$

for the equation *(IV.B.2:1)* , and

$$U_{i,j+1} = U_{i,j} + \frac{\varepsilon r}{2}[(U_{i+1,j} - 2U_{i,j} + U_{i-1,j}) + (U_{i+1,j+1} - 2U_{i,j+1} + U_{i-1,j+1})] -$$

$$\frac{rh}{4}[(U_{i+1,j} - U_{i-1,j}) + (U_{i+1,j+1} - U_{i-1,j+1})]\frac{(U_{i,j} + U_{i,j+1})}{2}$$

(IV.B.2:15)

for the equation *(IV.B.2:2)*. As can be seen the unknowns $U_{i-1,j+1}$, $U_{i,j+1}$ and $U_{i+1,j+1}$ are again related through a non-linear expression which has to be solved iteratively.

For the specific case of equation *(IV.B.2:2)* the non-linearity occurring stems from the approximation schemes due to the term $u\frac{\partial u}{\partial x}$,

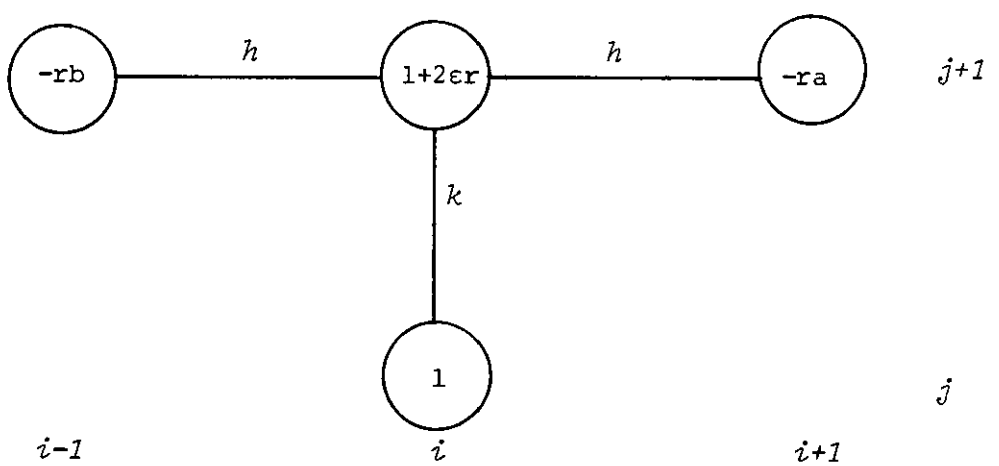


Figure IV.B.2-f2: The Molecular Diagram of the *Fully Implicit* Scheme for the Iterative Solution of Burgers' Equation.

but certainly, there are approximation alternatives in which this non-linearity can be avoided; for example, the non-linearity in scheme *(IV.B.2:15)* can be avoided if instead we use the *mean* of the *central-difference* approximations as,

$$u\frac{\partial u}{\partial x}\Big|_{i,j+\frac{1}{2}} \cong \frac{1}{4h}[(u_{i+1,j}-u_{i-1,j})u_{i,j+1}+(u_{i+1,j+1}-u_{i-1,j+1})u_{i,j}]. \qquad (IV.B.2:16)$$

The molecular diagram of formula *(IV.B.2:15)*, which is by the use of approximation *(IV.B.2:16)* and for illustrative purposes rewritten as,

$$-r(\frac{\varepsilon}{2}+\frac{h}{4}U_{i,j})U_{i-1,j+1}+[1+\varepsilon r+\frac{rh}{4}(U_{i+1,j}-U_{i-1,j})]U_{i,j+1}-r(\frac{\varepsilon}{2}-\frac{h}{4}U_{i,j})$$

$$U_{i+1,j+1} = \frac{r\varepsilon}{2}U_{i-1,j}+(1-\varepsilon r)U_{i,j}+\frac{r\varepsilon}{2}U_{i+1,j}\ , \qquad (IV.B.2:17)$$

is as in *Figure (IV.B.2-f3)*, where,

$$\left.\begin{aligned} &a_1 = b_1 = \frac{\varepsilon}{2}\ ,\quad c_1 = 1-\varepsilon r \\ &a_2 = \frac{\varepsilon}{2}-\frac{h}{4}U_{i,j}\ , c_2 = 1+\varepsilon r+\frac{rh}{4}(U_{i+1,j}-U_{i-1,j}) \\ &b_2 = \frac{\varepsilon}{2}+\frac{h}{4}U_{i,j} \end{aligned}\right\} \qquad (IV.B.2:18)$$

The popularity of the implicit methods is due mainly to their property of possessing unconditional stability, which leads to larger time-steps of integration and often increased accuracy. However, these methods are more expensive computationally, in comparison with the explicit methods, since they lead inevitably to the problem of

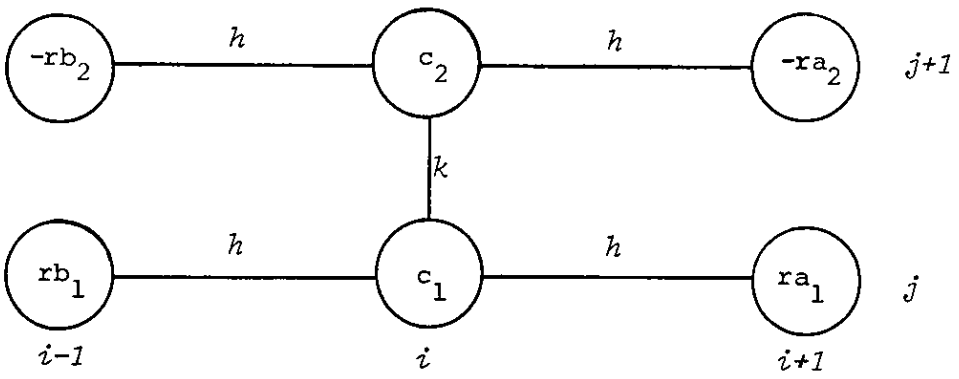


Figure IV.B.2-f3: The Molecular Diagram of the *Crank-Nicolson* Scheme for the Direct Solution of Burgers' Equation.

solving large numbers of linear systems of equations, i.e. tridiagonal, sparse quindiagonal, etc. On the other hand, with the implicit schemes, from parallel computational aspects, the unknowns are usually related through an expression and little or no independencies exist to be exploitable by parallel architectures. Consequently, with the increasing availability of parallel computers and their greater throughput, the explicit methods of solution not only offer simplicity, but also the capability that the solution can be obtained at every point *concurrently*. This important factor reinforces the need for improved explicit procedures for utilization on parallel computers.

Finally, all the various conventional approximation schemes discussed herein can be obtained from the generalized approximation (see Abdullah [*ABDU83*]),

$$U_{i,j+1} = U_{i,j} + kG\left[ih, (j+\tfrac{1}{2})k, \frac{U_{i,j}+U_{i,j+1}}{2}, \frac{\alpha_1\Delta_x U_{i,j+1} + \alpha_2\nabla_x U_{i,j} + \alpha_1'\nabla_x U_{i,j+1} + \alpha_2'\Delta_x U_{i,j}}{2h}, \frac{1}{h^2}(\theta_1\delta_x U_{i+\frac{1}{2},j+1} - \theta_2\delta_x U_{i-\frac{1}{2},j+1}) + \frac{1}{h^2}(\theta_1'\delta_x U_{i+\frac{1}{2},j} - \theta_2'\delta_x U_{i-\frac{1}{2},j})\right] \qquad (IV.B.2:19)$$

for the equation *(IV.B.2:1)*, and

$$U_{i,j+1} = U_{i,j} + \varepsilon r(\theta_1\delta_x U_{i+\frac{1}{2},j+1} - \theta_2\delta_x U_{i-\frac{1}{2},j+1} + \theta_1'\delta_x U_{i+\frac{1}{2},j} - \theta_2'\delta_x U_{i-\frac{1}{2},j}) - k\,\frac{(U_{i,j}+U_{i,j+1})}{2}\,\frac{(\alpha_1\Delta_x U_{i,j+1} + \alpha_2\nabla_x U_{i,j} + \alpha_1'\nabla_x U_{i,j+1} + \alpha_2'\Delta_x U_{i,j})}{2h} \qquad (IV.B.2:20)$$

for the equation *(IV.B.2:2)*. The parameters θ's and α's have to satisfy the compulsory conditions,

$$\left.\begin{aligned} &\sum_{l=1}^{2} (\theta_l+\theta'_l) = 2 \\ &\sum_{l=1}^{2} (\alpha_l+\alpha'_l) = 2 \\ &-\theta_1+\theta_2-\theta'_1+\theta'_2 = 0 \end{aligned}\right\} , \qquad (IV.B.2:21)$$

while the operators δ_x, Δ_x and ∇_x are the *central-*, *forward-* and *backward-difference* operators[†] with respect to the x variable, respectively.

[†] *Namely,* $\delta_x U_{i,j} = U_{i+\frac{1}{2},j} - U_{i-\frac{1}{2},j}$, $\Delta_x U_{i,j} = U_{i+1,j} - U_{i,j}$, $\nabla_x U_{i,j} = U_{i,j} - U_{i-1,j}$.

IV.B.3: THE NEW CLASS OF 'GROUP EXPLICIT' - *GE* SOLUTION METHODS

In a summarizing prologue, a new solution strategy is introduced herein and has been extensively experimented on parallel systems in the remainder of this *Chapter*; it combines stable asymmetric approximations to the p.d.e.'s, which, when coupled in groups of $2^{\dagger}$ adjacent points on the grid, result in implicit equations which are easily convertible to explicit form. In particular, by judicious use of alternating this strategy on the grid points of the domain results in new explicit algorithms which possess unconditional stability. The merit of these approaches results in accurate solutions because of truncation error cancellations.

More analytically, let us consider the generalized approximations *(IV.B.2:19,20)* and the following choices for θ'*s* and α'*s*:

(i) $\theta_1 = \theta_2' = 1,\ \theta_2 = \theta_1' = 0,\ \alpha_1 = \alpha_2 = 1,\ \alpha_1' = \alpha_2' = 0$; and,

(ii) $\theta_1 = \theta_2' = 0,\ \theta_2 = \theta_1' = 1,\ \alpha_1 = \alpha_2 = 0,\ \alpha_1' = \alpha_2' = 1.$

The former choice will result in

$$U_{i,j+1} = U_{i,j} + kG\left[ih, (j+\tfrac{1}{2})k, \frac{U_{i,j}+U_{i,j+1}}{2}, \frac{(U_{i+1,j+1}-U_{i,j+1})+(U_{i,j}-U_{i-1,j})}{2h}, \frac{1}{h^2}(U_{i+1,j+1}-U_{i,j+1}-U_{i,j}+U_{i-1,j})\right], \qquad \textit{(IV.B.3:1)}$$

and

$$-raU_{i+1,j+1} + (1+ra)U_{i,j+1} = (1-rb)U_{i,j} + rbU_{i-1,j}, \qquad \textit{(IV.B.3:2)}$$

respectively; while the latter choice will result in

$$U_{i,j+1} = U_{i,j} + kG\left[ih, (j+\tfrac{1}{2})k, \frac{U_{i,j}+U_{i,j+1}}{2}, \frac{(U_{i,j+1}-U_{i-1,j+1})+(U_{i+1,j}-U_{i,j})}{2h}, \frac{1}{h^2}(U_{i+1,j}-U_{i,j}-U_{i,j+1}+U_{i-1,j+1})\right], \qquad \textit{(IV.B.3:3)}$$

---

†*Four points for 2-dimensions.*

and

$$-rbU_{\imath-1,j+1}+(1+rb)U_{\imath,j+1} = (1-ra)U_{\imath,j}+raU_{\imath+1,j}, \qquad (IV.B.3:4)$$

respectively, where for both choices

$$\left.\begin{aligned} a &= \varepsilon - \frac{h}{4}(U_{\imath,j+1}+U_{i,j}) \\ b &= \varepsilon + \frac{h}{4}(U_{\imath,j+1}+U_{i,j}) \end{aligned}\right\} . \qquad (IV.B.3:5)$$

These asymmetric formulae, whose molecular diagrams for the particular case of Burgers' equation are as in *Figures (IV.B.3-f1,f2)*, respectively, are due to Saul'yev [*SAUL64*]; they are unconditionally stable for $r>0$ and *semi-explicit* in the sense that if the equations *(IV.B.3:1,2)* are solved in a 'right-to-left' - (*RL*) direction and the equations *(IV.B.3:3,4)* in a 'left-to-right' - (*LR*) direction, then the need to solve a linear system for the solution on each line is averted. In general, the equations *(IV.B.3:1,2)* are non-linear in $U_{i,j+1}$ and $U_{i+1,j+1}$, while the equations *(IV.B.3:3,4)* are non-linear in $U_{i,j+1}$ and $U_{i-1,j+1}$. The equations *(IV.B.3:1,3)* can be neatly described as,

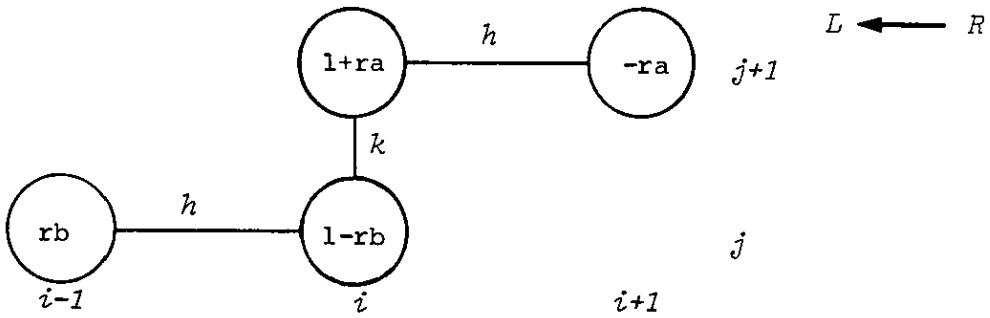


Figure IV.B.3-f1: The Molecular Diagram of Saul'yev's Asymmetric Formula *(IV.B.3:2)*.

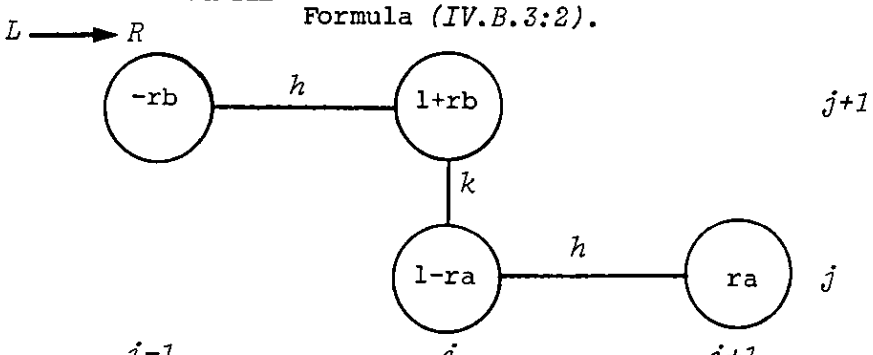


Figure IV.B.3-f2: The Molecular Diagram of Saul'yev's Asymmetric Formula *(IV.B.3:4)*.

$$a_1 U_{i,j+1} + b_1 U_{i+1,j+1} = G_1(U_{i,j}, U_{i-1,j}) \qquad (IV.B.3:6)$$

and

$$a_2 U_{i,j+1} + b_2 U_{i-1,j+1} = G_2(U_{i,j}, U_{i+1,j}), \qquad (IV.B.3:7)$$

respectively, where a_1, b_1 are functions of $U_{i,j+1}$ and $U_{i+1,j+1}$, and a_2, b_2 are functions of $U_{i,j+1}$ and $U_{i-1,j+1}$. In particular for equations *(IV.B.3:2,4)*, they can be linearized if the values of **a,b** in *(IV.B.3:5)* are re-defined as,

$$\left.\begin{aligned} a' &= \varepsilon - \frac{h}{2} U_{i,j} \\ b' &= \varepsilon + \frac{h}{2} U_{i,j} \end{aligned}\right\} . \qquad (IV.B.3:8)$$

The equations *(IV.B.3:1,2,3,4)* are *ladder-step* formulae and therefore algorithms similar to those suggested by Larkin [*LARK64*] for the *heat-conduction* equation are possible. More explanatorily, these algorithms are:

(i) - By use of equations *(IV.B.3:6,2)* from a *RL* - direction explicitly as,

$$U_{i,j+1}^{(n+1)} = \frac{1}{a_1^{(n)}} \left\{ G_1(U_{i,j}, U_{i-1,j}) - b_1^{(n)} U_{i+1,j+1}^{(n+1)} \right\}, \qquad (IV.B.3:9)$$

and

$$U_{i,j+1}^{(n+1)} = \frac{1}{(1+ra^{(n)})} \left\{ ra^{(n)} U_{i+1,j+1}^{(n+1)} + (1-rb^{(n)}) U_{i,j} + rb^{(n)} U_{i-1,j} \right\}, \qquad (IV.B.3:10)$$

respectively, where the superscript n refers to the n^{th} iteration number.

(ii) - By use of equations *(IV.B.3:7,4)* from a *LR* - direction explicitly as,

$$U_{i,j+1}^{(n+1)} = \frac{1}{a_2^{(n)}} \left\{ G_2(U_{i,j}, U_{i+1,j}) - b_2^{(n)} U_{i-1,j+1}^{(n+1)} \right\}, \qquad (IV.B.3:11)$$

and

$$U_{i,j+1}^{(n+1)} = \frac{1}{(1+rb^{(n)})} \left\{ rb^{(n)} U_{i-1,j+1}^{(n+1)} + (1-ra^{(n)}) U_{i,j} + ra^{(n)} U_{i+1,j} \right\}, \qquad (IV.B.3:12)$$

respectively.

(iii) - By use of equations *(IV.B.3:9,10)* at the j^{th} time-level from a *RL* - direction and alternatively using equations *(IV.B.3:11,12)*, respectively, at the $(j+1)^{th}$ time-level from a *LR* - direction (i.e. a *RL-LR* combination).

(iv) - By use of equations *(IV.B.3:9,10)* as in algorithm *(i)* and equations *(IV.B.3:11,12)* as in algorithm *(ii)*, at each time-level, and then average the results for the final answer for that level. This averaging approach has great merit compared to the others because of truncation error cancellations in the combined solutions.

The last two algorithms are *semi-explicit* in nature and certainly less preferable in comparison with the pure explicit scheme which is normally easier to handle irrespective of the computational work involved.

Recently, an interesting new variation of the use of Saul'yev's asymmetric equations was investigated by Evans and Abdullah [*EVAN83b*]. The central theme of the idea was not to restrict the use of those equations solely along the x lines in the *RL* - and *LR* - directions, but to apply them to groups of 2 points successively along every line in the manner illustrated in *Figure (IV.B.3-f3)*; the symbol ○ denotes the use of equations *(IV.B.3:9,10)* and the symbol □ the use of equations *(IV.B.3:11,12)*.

As we have already mentioned, the explicit methods although very suitable for parallel processing always deny us reasonable accuracy and some stability; on the other hand, the implicit schemes offer stability, but the exploitation of these methods for parallel processing may be difficult and possibly inefficient. The semi-explicit algorithms discussed above enable us with a trade-off between stability and the

possibility of them being suitable for implementation on parallel systems. Furthermore, it is possible to express the semi-explicit schemes in terms of pure explicit formulae to enable their efficient implementation. Of this sort is the class of 'Group Explicit' - *GE* methods, which, to some extent, provide more effective formulae for implementation in parallel. In fact, the coupled use of Saul'yev's asymmetric equations at the points $(i,j+1)$ and $(i+1,j+1)$ results in a (2×2)-set of implicit finite-difference equations, which can be easily converted to explicit form as is developed in the following.

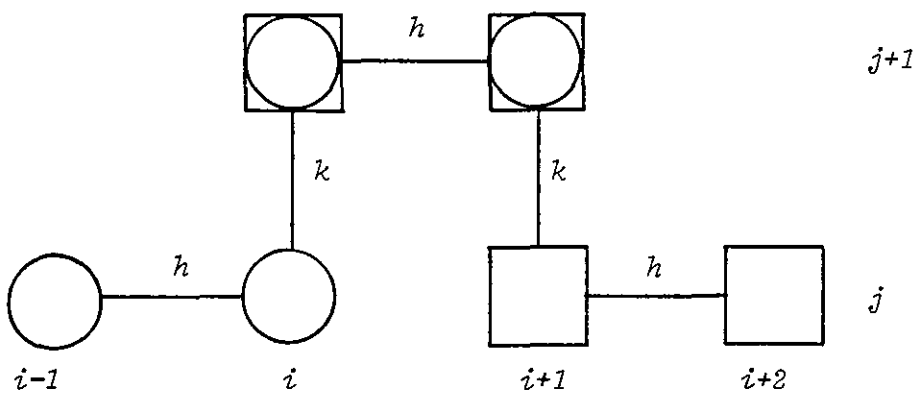


Figure IV.B.3-f3: A Variation of the Use of Saul'yev's Asymmetric Equations.

In order to formulate the *GE* equations, we assume, without any loss of generality, that the line segment $0\leq x\leq1$ is divided into an *even* number m of equal sub-intervals, which implies that at every time-level the number of internal points is *odd*, i.e. $(m-1)$. Now consider any group of two points, i.e. $(i,j+\frac{1}{2})$ and $(i+1,j+\frac{1}{2})$, and use equations *(IV.B.3:6,2)* at the point $(i,j+\frac{1}{2})$ and equations *(IV.B.3:7,4)* at the point $(i+1,j+\frac{1}{2})$, correspondingly grouped together, to give, in matrix form, the (2×2) systems of equations,

$$\begin{bmatrix} a_1^{(n)} & b_1^{(n)} \\ b_2^{(n)} & a_2^{(n)} \end{bmatrix} \begin{bmatrix} U_{i,j+1}^{(n+1)} \\ U_{i+1,j+1}^{(n+1)} \end{bmatrix} = \begin{bmatrix} G_1(U_{i,j}, U_{i-1,j}) \\ G_2(U_{i+1,j}, U_{i+2,j}) \end{bmatrix} \quad (IV.B.3:13)$$

and

$$\begin{bmatrix} 1+\hat{a}_1^{(n)}r & -\hat{a}_1^{(n)}r \\ -\hat{b}_2^{(n)}r & 1+\hat{b}_2^{(n)}r \end{bmatrix} \begin{bmatrix} U_{i,j+1}^{(n+1)} \\ U_{i+1,j+1}^{(n+1)} \end{bmatrix} = \begin{bmatrix} 1-\hat{b}_1^{(n)}r & 0 \\ 0 & 1-\hat{a}_2^{(n)}r \end{bmatrix} \begin{bmatrix} U_{i,j} \\ U_{i+1,j} \end{bmatrix} + \begin{bmatrix} \hat{b}_1^{(n)}rU_{i-1,j} \\ \hat{a}_2^{(n)}rU_{i+2,j} \end{bmatrix}, \quad (IV.B.3:14)$$

where,

$$\hat{a}_1^{(n)} = \varepsilon - \frac{h}{4}(U_{i,j+1}^{(n)} + U_{i,j}); \quad \hat{a}_2^{(n)} = \varepsilon - \frac{h}{4}(U_{i+1,j+1}^{(n)} + U_{i+1,j})$$
$$\hat{b}_1^{(n)} = \varepsilon + \frac{h}{4}(U_{i,j+1}^{(n)} + U_{i,j}); \quad \hat{b}_2^{(n)} = \varepsilon + \frac{h}{4}(U_{i+1,j+1}^{(n)} + U_{i+1,j}). \quad (IV.B.3:15)$$

In particular for the (2×2) system of equations *(IV.B.3:14)*, with the explicit form of which we shall extensively experiment, it can be linearized, similarly to *(IV.B.3:8)*, if the values of $\hat{a}_1, \hat{a}_2, \hat{b}_1, \hat{b}_2$ are re-defined as,

$$\hat{a}_1 = \varepsilon - \frac{h}{2}U_{i,j}; \qquad \hat{a}_2 = \varepsilon - \frac{h}{2}U_{i+1,j}$$
$$\hat{b}_1 = \varepsilon + \frac{h}{2}U_{i,j}; \qquad \hat{b}_2 = \varepsilon + \frac{h}{2}U_{i+1,j}. \quad (IV.B.3:16)$$

As we shall see later on in the corresponding paragraphs, despite the small time-step values k the results at each time-level were so accurate that there was no need for any iteration of the non-linear problem and the use of such values for $\hat{a}_1, \hat{a}_2, \hat{b}_1, \hat{b}_2$ as in *(IV.B.3:16)* was absolutely justified.

The explicit form of the (2×2) systems of equations *(IV.B.3:13,14)*

is given by,

$$\begin{bmatrix} U_{1,j+1}^{(n+1)} \\ U_{1+1,j+1}^{(n+1)} \end{bmatrix} = \frac{1}{\Delta_1} \begin{bmatrix} a_2^{(n)} & -b_1^{(n)} \\ -b_2^{(n)} & a_1^{(n)} \end{bmatrix} \begin{bmatrix} G_1(U_{1,j}, U_{i-1,j}) \\ G_2(U_{1+1,j}, U_{1+2,j}) \end{bmatrix} \qquad (IV.B.3:17)$$

and

$$\begin{bmatrix} U_{1,j+1}^{(n+1)} \\ U_{1+1,j+1}^{(n+1)} \end{bmatrix} = \frac{1}{\Delta_2} \left\{ \begin{bmatrix} (1+\hat{b}_2^{(n)}r-\hat{b}_1^{(n)}r-\hat{b}_1^{(n)}\hat{b}_2^{(n)}r^2) & (1-\hat{a}_2^{(n)}r)\hat{a}_1^{(n)}r \\ \hat{b}_2^{(n)}r(1-\hat{b}_1^{(n)}r) & (1+\hat{a}_1^{(n)}r-\hat{a}_2^{(n)}r-\hat{a}_1^{(n)}\hat{a}_2^{(n)}r^2) \end{bmatrix} \begin{bmatrix} U_{1,j} \\ U_{i+1,j} \end{bmatrix} + \begin{bmatrix} (1+\hat{b}_2^{(n)}r)\hat{b}_1^{(n)}rU_{1-1,j} + \hat{a}_1^{(n)}\hat{a}_2^{(n)}r^2U_{1+2,j} \\ \hat{b}_1^{(n)}\hat{b}_2^{(n)}r^2U_{1-1,j} + (1+\hat{a}_1^{(n)}r)\hat{a}_2^{(n)}rU_{1+2,j} \end{bmatrix} \right\},$$

$$(IV.B.3:18)$$

where the *inverse* of the coefficient matrices for $U_{i,j+1}, U_{i+1,j+1}$ has been estimated through the use of their *adjoint* matrices and corresponding determinants,

$$\Delta_1 = a_1^{(n)}a_2^{(n)} - b_1^{(n)}b_2^{(n)} \quad ; \quad \Delta_2 = 1+\hat{a}_1^{(n)}r+\hat{b}_2^{(n)}r \ . \qquad (IV.B.3:19)$$

The equations *(IV.B.3:17,18)* are the explicit equations which are computationally easier to handle. The molecular diagrams of the *GE* formulae, for the particular case of Burgers' equation *(IV.B.2:2)*, are as in *Figure (IV.B.3-f4)*; whilst for the ungrouped (single) points near the *right* and *left* boundaries we use Saul'yev's equations *(IV.B.3:10,12)*, respectively.

Now making use of these *GE* equations we shall consider a variety of schemes for this class of methods, which can be at first established obeying the previous assumption for an *even* number of equal sub-intervals.

In accordance with this assumption, for a later practical matching, the equations *(IV.B.3:10,12)*, in specific, can be correspondingly rewritten as, (for the *right* near-boundary point):

$$U_{m-1,j+1}^{(n+1)} = \frac{1}{(1+ra^{(n)})}\left\{ra^{(n)}U_{m,j+1}^{(n+1)}+(1-rb^{(n)})U_{m-1,j}+rb^{(n)}U_{m-2,j}\right\}, \quad \textit{(IV.B.3:20)}$$

(for the *left* near-boundary point):

$$U_{1,j+1}^{(n+1)} = \frac{1}{(1+rb^{(n)})}\left\{rb^{(n)}U_{0,j+1}^{(n+1)}+(1-ra^{(n)})U_{1,j}+ra^{(n)}U_{2,j}\right\}. \quad \textit{(IV.B.3:21)}$$

(i) - The '*G*roup *E*xplicit with *R*ight ungrouped point' - *GER* scheme

This scheme is obtained by either use of the systems of equations *(IV.B.3:17,18)* for $\frac{1}{2}(m-2)$ times, for the first $(m-2)$ points grouped *2* at a time and the corresponding use of either of equations *(IV.B.3:9,20)* for the last, i.e. $(m-1)^{th}$, right ungrouped point, at every time-level. In implicit matrix form, for the particular case of Burgers' equation,

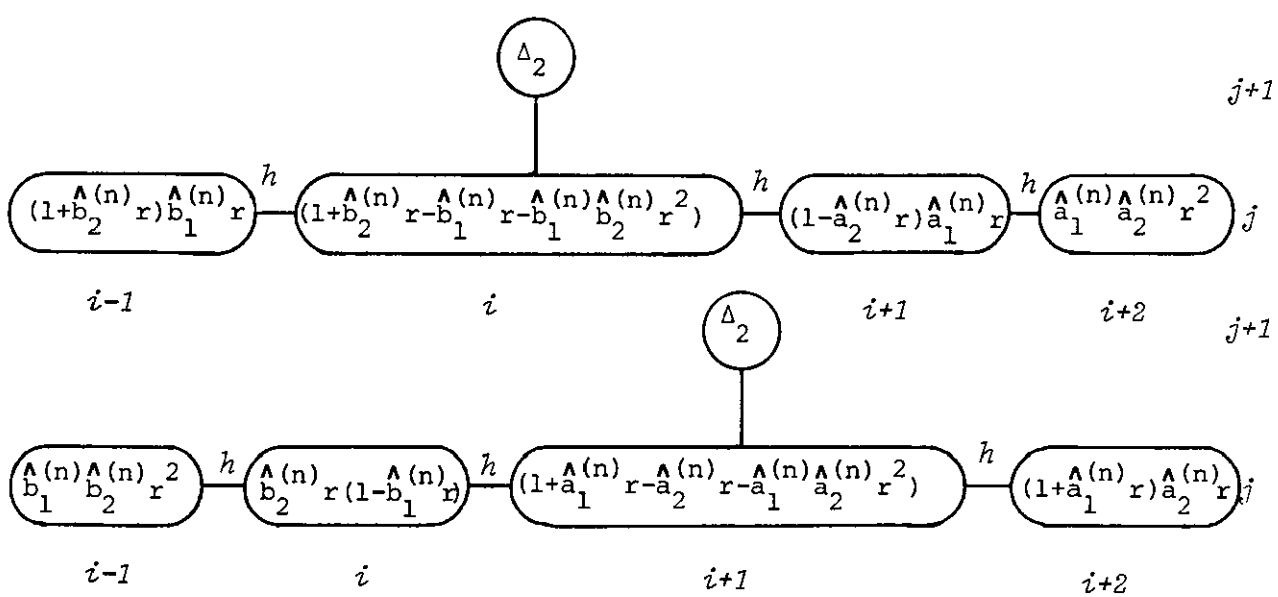
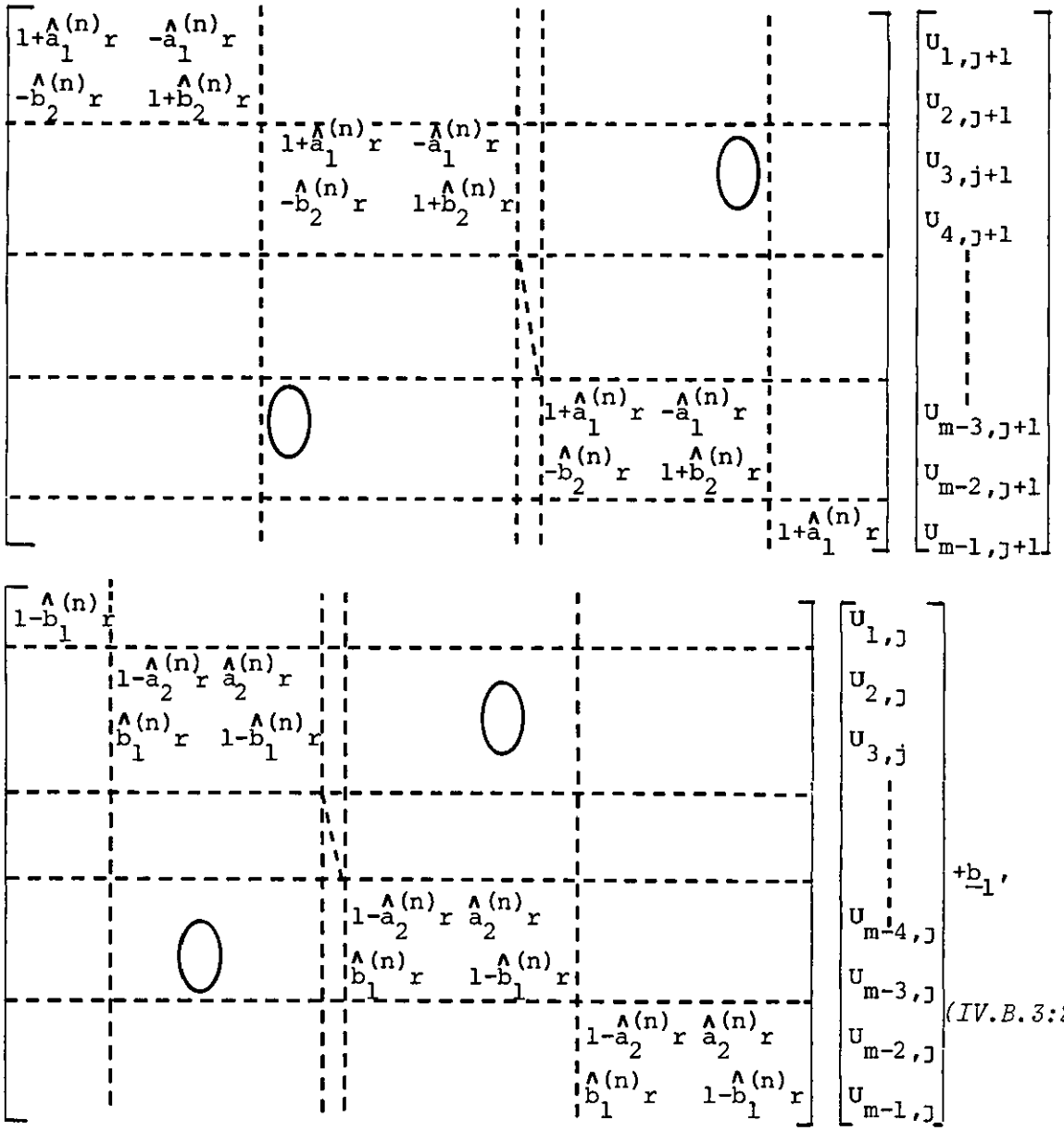


Figure IV.B.3-f4: The Molecular Diagrams of the *GE* Formulae for Burgers' Equation.

it is given by,



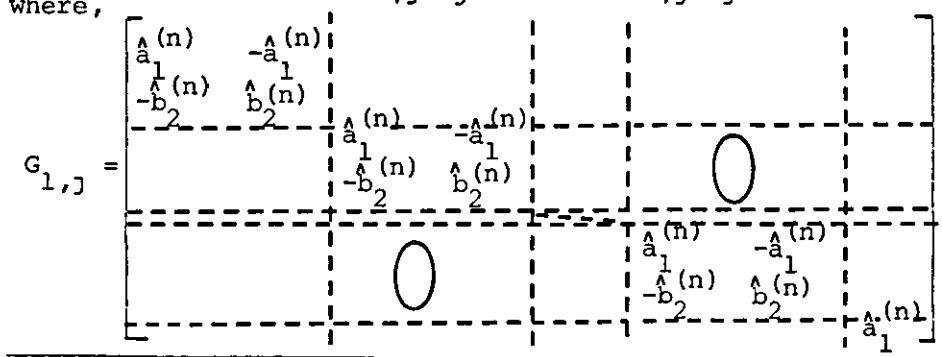
where, $\hat{a}_1, \hat{a}_2, \hat{b}_1, \hat{b}_2$ are as defined in *(IV.B.3:15)*, and

$$\underline{b}_1^{T^\dagger} = [\hat{b}_1^{(n)} r U_{0,j}, 0, \ldots, 0, \hat{a}_1^{(n)} r U_{m,j+1}] \qquad \textit{(IV.B.3:23)}$$

consisting of known boundary values; or, otherwise,

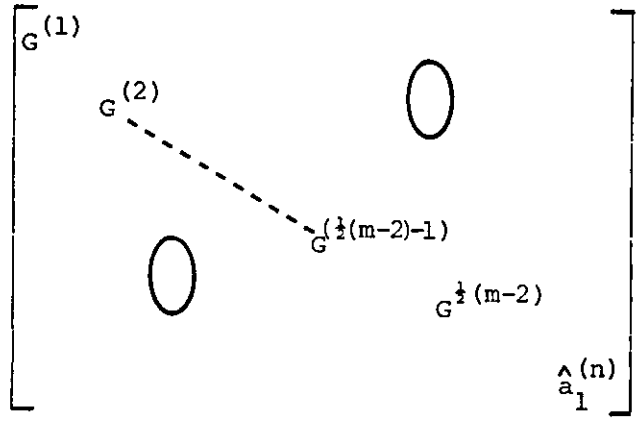
$$(I^\ddagger + rG_{1,j})\underline{U}_{j+1} = (I^\ddagger - rG_{2,j})\underline{U}_j + \underline{b}_1 \,, \qquad \textit{(IV.B.3:24)}$$

where,



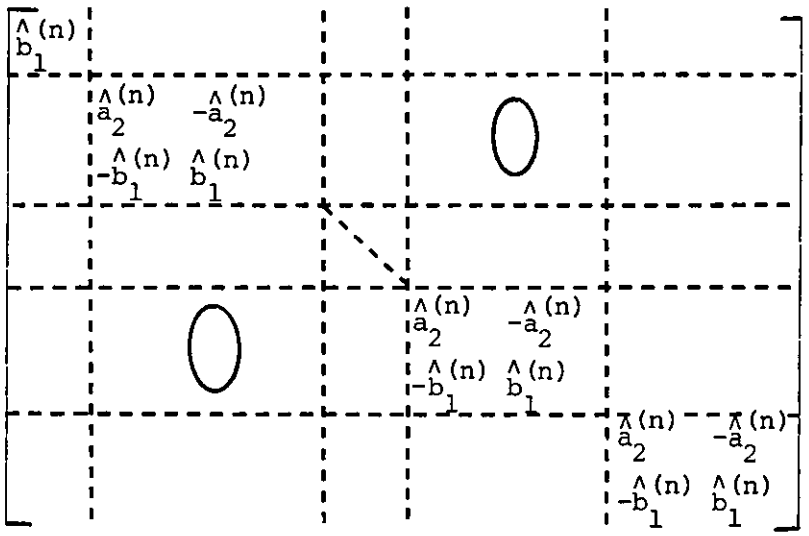
†*The 'Transpose' of* $\underline{b}_1$.

‡*The 'Identity' matrix.*

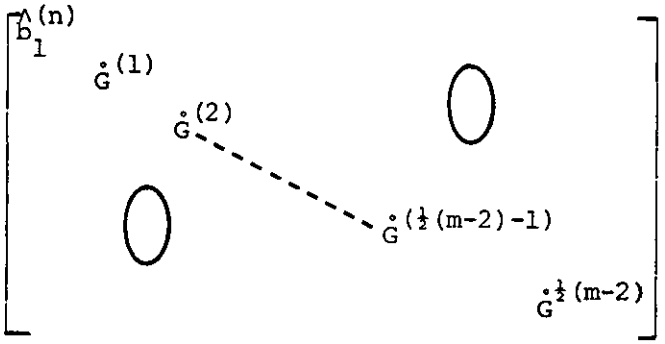


(IV.B.3:25)

and



$G_{2,j} =$



, (IV.B.3:26)

with,

$$G^{(\imath)} = \begin{bmatrix} \hat{a}_1^{(n)} & -\hat{a}_1^{(n)} \\ -\hat{b}_2^{(n)} & \hat{b}_2^{(n)} \end{bmatrix} \qquad \text{for } \imath=1,2,\ldots,\tfrac{1}{2}(m-2),$$

$$\dot{G}^{(\imath)} = \begin{bmatrix} \hat{a}_2^{(n)} & -\hat{a}_2^{(n)} \\ -\hat{b}_1^{(n)} & \hat{b}_1^{(n)} \end{bmatrix} \qquad j=0,1,2,\ldots \; . \qquad \text{(IV.B.3:27)}$$

This scheme can be illustrated by the *brick* diagram in *Figure (IV.B.3-f5,i)*.

(ii) - The 'Group Explicit with Left ungrouped point' - *GEL* scheme

This scheme is obtained by either use of equations *(IV.B.3:11,21)* for the 1^{st} left ungrouped point and $\frac{1}{2}(m-2)$ times the corresponding use of either of the systems of equations *(IV.B.3:17,18)* for the remaining pairs of points, at every time-level. In accordance with the above definition of matrices $G_{1,j}, G_{2,j}$, similarly as before, for the particular case of Burgers' equation, it is given by the formula,

$$(I+rG_{2,j})\underline{U}_{j+1} = (I-rG_{1,j})\underline{U}_j + \underline{b}_2 \ , \qquad \textit{(IV.B.3:28)}$$

where,

$$\underline{b}_2^T = [\hat{b}_1^{(n)} rU_{0,j+1}, 0, \ldots, 0, \hat{a}_1^{(n)} rU_{m,j}] \ ; \qquad \textit{(IV.B.3:29)}$$

the *brick* diagram for this scheme is as in *Figure (IV.B.3-f5,ii)*.

(iii) - The '(Single) Alternating Group Explicit' - *(S)AGE* scheme

Another variation involving the coupled use of the *GER* and *GEL* schemes at every alternate time-level. In a similar manner as above, for the particular case of Burgers' equation, it is given by the formulae,

$$\left.\begin{aligned} (I+rG_{1,j})\underline{U}_{j+1} &= (I-rG_{2,j})\underline{U}_j + \underline{b}_1 \\ (I+rG_{2,j})\underline{U}_{j+2} &= (I-rG_{1,j})\underline{U}_{j+1} + \underline{b}_2 \end{aligned}\right\} \ ; \qquad \textit{(IV.B.3:30)}$$

the *brick* diagram for this scheme is as in *Figure (IV.B.3-f5,iii)*.

(iv) - The '(Double) Alternating Group Explicit' - *(D)AGE* scheme

This variation has been developed from a periodic rotation of the previous two time-level *(S)AGE* scheme, resulting in a *four* time-level step process with the second half cycle in reverse order. Consequently, again for the particular case of Burgers' equation, it is given by the formulae,

$$\left.\begin{aligned}
(I+rG_{1,j})\underline{U}_{j+1} &= (I-rG_{2,j})\underline{U}_{j}+\underline{b}_{1} \\
(I+rG_{2,j})\underline{U}_{j+2} &= (I-rG_{1,j})\underline{U}_{j+1}+\underline{b}_{2} \\
(I+rG_{2,j})\underline{U}_{j+3} &= (I-rG_{1,j})\underline{U}_{j+2}+\underline{b}_{2} \\
(I+rG_{1,j})\underline{U}_{j+4} &= (I-rG_{2,j})\underline{U}_{j+3}+\underline{b}_{1}
\end{aligned}\right\} ; \qquad (IV.B.3:31)$$

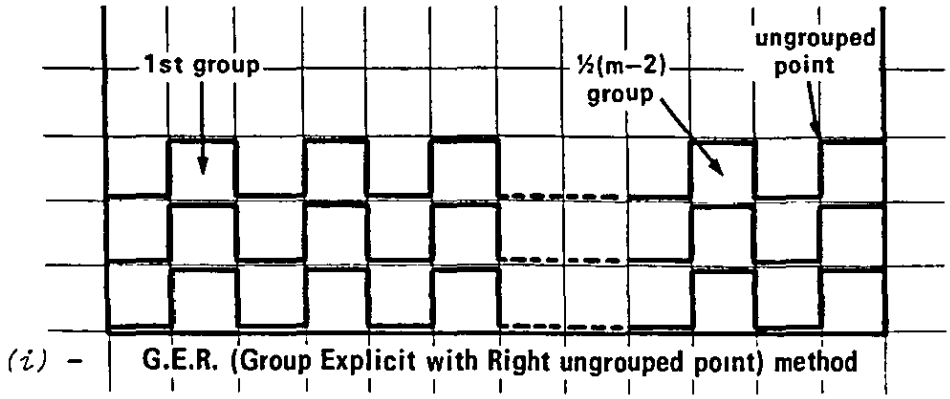
this scheme is represented by the *brick* diagram in *Figure (IV.B.3-f5,iv)*.

The estimate of the truncation errors of all the schemes in this class can be shown by Taylor's series expansion to be of order $O(k+h^2+k/h)$. These schemes will be consistent to the original problem if $k/h \to 0$ and when $k \to 0$ and $h \to 0$.

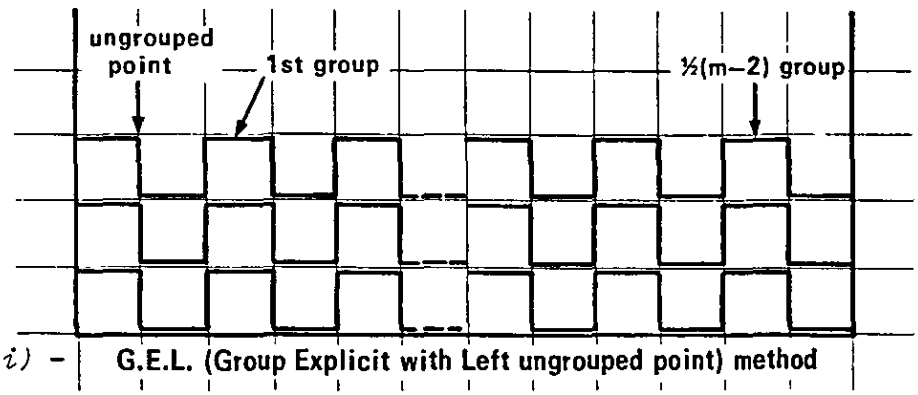
From the aspect of the stability analysis, it has been considered, using the matrix method (see Evans and Abdullah [*EVAN83*]), for the specific case of Burgers' equation only, since such an analysis for solving the general non-linear parabolic equation *(IV.B.2:1)* would be very complicated and difficult. In fact, even in this case, since $G_{1,j}$, $G_{2,j}$ are not 'commutative' and furthermore they are matrices with variable elements, where the verification of the 'positive definite' property is not obviously straightforward, the analysis of stability of the *GE* type of schemes is very complicated. However, since these schemes are derived from stable semi-explicit schemes, the probability of them being stable is very high. The stability of the *GE* schemes can also be seen from the numerical results exhibited in the following paragraphs; therein, the *GE* equations have been considered under the assumption of an *odd* number of intervals, which results in slightly different schemes, much more balanced and computationally preferable, thus integrating the image of this new powerful class of explicit methods.

Finally, to prologue the following paragraph, the Standard Explicit method is primarily implemented and analyzed on the *NEPTUNE* prototype,

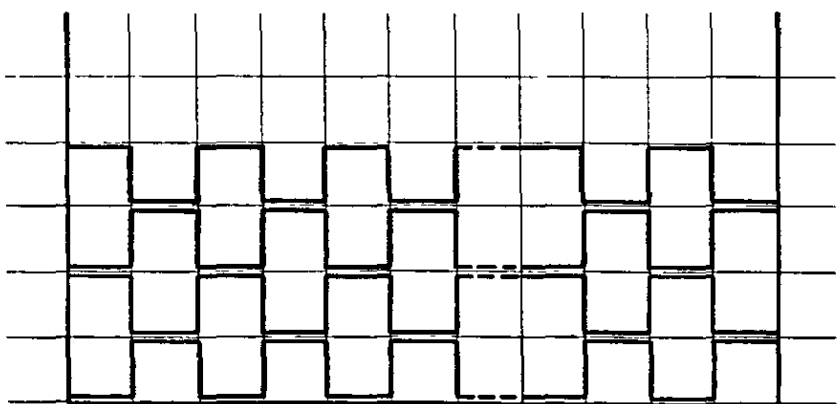
Even Number of Intervals



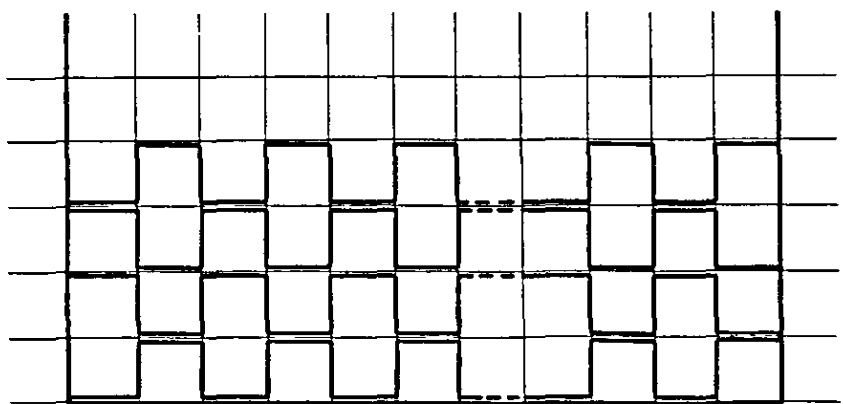
(i) – G.E.R. (Group Explicit with Right ungrouped point) method



(ii) – G.E.L. (Group Explicit with Left ungrouped point) method



(iii) – S.A.G.E. (Single Alternating Group Explicit) method



(iv) – D.A.G.E. (Double Alternating Group Explicit) method

Figure IV.B.3-f5: The Representative Diagrams of the Various *GE* Schemes.

since it will form the basis for performance comparisons with the *GE* schemes which will be fully and individually exploited on this system in the subsequent paragraphs.

IV.B.3.1: The Standard Explicit Method: Performance Model, Experimental Results And Performance Analysis On The *'NEPTUNE'* Prototype System

After having presented the basic concepts of the various finite-difference solution schemes, we proceed with the parallel implementation of the Standard Explicit method represented by the formula

$$U_{i,j+1} = \varepsilon r(U_{i+1,j}-2U_{i,j}+U_{i-1,j})+[1-\frac{rh}{2}(U_{i+1,j}-U_{i-1,j})]U_{i,j}, \qquad (IV.B.3.1:1)$$

for $r=k/h^2$, since the performance analysis results obtained from this method will be directly compared with the corresponding results obtained from the new *GE* methods. For our numerical experiments with Burgers' equation *(IV.B.2:2)* the following exact solution has been chosen (see Madsen and Sincovec [*MADS76*]),

$$u(x,t) = \frac{0.1e^{-A}+0.5e^{-B}+e^{-C}}{e^{-A}+e^{-B}+e^{-C}}, \quad 0\leq x\leq 1,\ t\geq 0 \qquad (IV.B.3.1:2)$$

where,

$$\left.\begin{aligned} A &= \frac{0.05}{\varepsilon}(x-0.5+4.95t) \\ B &= \frac{0.25}{\varepsilon}(x-0.5+0.75t) \\ C &= \frac{0.5}{\varepsilon}(x-0.375) \end{aligned}\right\}. \qquad (IV.B.3.1:3)$$

This problem in practice would have a very small value of ε and this in turn means that the interval [0,1] in the x direction has to be subdivided into a very fine grid in order to obtain reasonable accuracy.

In our experimental work, for a reasonably small value of ε, the Standard Explicit method, as well as the following *GE* methods, has been analytically tested for a wide range of internal points and time-levels reaching the extremest figures allowed by the *NEPTUNE* system; in particular, the latter's number is adjusted in accordance with r to maintain the same overall time-advance length. In all cases, despite the very small values of the time-step k, the results at each time-level are so accurate that there is no need for any iteration of the non-linear problem.

From the aspect of parallel programming, the previous formula *(IV.B.3.1:1)* has been implemented in an unavoidable, but highly efficient, *synchronized* manner which is due to the nature of the method; *asynchronous* algorithms, in fact, are the oldest and most primitive ways of implementation, since as the methods and problems became more and more complicated, synchronization activities were inevitably brought into use. Finally, the number of sub-intervals m is assumed to be *odd*, which implies that at every time-level the number of internal points is *even*, i.e. $(m-1)$.

Prior to proceeding with the discussion of the inherent parallelism of the method and the actual experimentation and analysis of the obtained results, and although we have already provided our performance prediction framework (see *par.-II.B.3.1*), we ought to make, for the interested reader, an essential clarification concerning the style of the performance analysis supported by some other different 'schools' of analysis.

Since quite a few such experimental systems, otherwise called 'Multiprocessor testbeds', have been built to investigate algorithm performance, a comprehensive approach to performance prediction requires

accurate performance frameworks in order to reduce the overall experimentation time. In accordance, most of the Multiprocessor performance frameworks have been theoretically based on statistical methods, predicting statistical mean values for performance over some time interval.

In particular, a 'school' of performance investigation (see Baudet [*BAUD78*]) performs an analysis of algorithms employing simple techniques on order statistics and queueing theory, while others (see Robinson [*ROBI79*]) approach the analysis† through the rules of probability theory. In particular for the former 'school', it is only concerned with the investigation of the inherent parallelism, of the application in hand, in terms of the total run-time.

In general, these approaches may give a good approximate estimate to the experimental timing results, and this can be very useful theoretically since it can be used to predict the *optimal decomposition* of a problem (i.e. the optimal number of processes to create in order to, for example, minimize the overall execution time). In other words, a successful implementation of these approaches can be proved as a theoretical guide-line of high assistance in answering the important question that concerns the best combination amongst the various parallel computer architectures and parallel algorithms in order to solve, in the most efficient way, a specific problem.

However, in real-time shared memory based systems the situation is quite often different, since individual systems may involve such specialized hardware and software features that make them more enhanced than others and thus of not accurately predictable performance via the

†*Mainly for sorting and merging algorithms.*

framework of such theoretical approaches. These cases are easily taken care of in our 'school' of deterministic performance analysis, which views performance as the interaction of resources demanded by programs and provided by the Multiprocessor system, in both system dependent and independent manner; it also matches algorithm to machine trying to avoid the danger of rejecting an algorithm because it performs badly on one particular parallel prototype system. Hence, our framework of analysis attempts a more realistic and direct approach, being, nearly from every aspect, *complete*, *accurate*[†] and *credible*, exclusively referring, without any loss of generality though, to the specific system in hand, the *NEPTUNE* prototype; and that is because, although we have designed general, in nature, parallel algorithms for *MIMD* complexes of processors, the attempt to exploit as much as it was possible *NEPTUNE*'s hardware potential and resource provisions has significantly affected the *'thinking'* in our programming.

A similar framework to ours has been set by Vrsalovic, et al [*VRSA84*], who presented a model for predicting Multiprocessor performance on iterative algorithms; these algorithms are made up of repeated *application cycles* consisting of some synchronous or asynchronous amount of accesses to global data and local processing. In particular, they are using the notion of *cyclic processing power* defined as the effective number of processors (that is the number of processors not 'idle' due to contention), working cooperatively in every cycle.

Returning to the *NEPTUNE* prototype system, let us discuss some absolutely necessary specialized characteristics of it that will be mainly and repeatedly utilized in our analysis, in combination with

[†]*Under the condition that the values of all system parametric figures are accurately provided.*

the resource provisions *Table* presented in *(Appendix C-II/par.-II.B.3.1)*. More specifically, two subroutines are available for obtaining the timing information. The routines should be embedded within a *$DOALL/ $PAREND* sequence to force each processor to execute them. The timing is started or restarted with

CALL TIMEST

and current timing(s) is obtained using

CALL TIMOUT(ITIME),

where *ITIME* must be declared as a *shared* array of size *100* and printed out from a subsequent sequential path, its results being arranged in *8* columns.

The timing results for each processor are held in *ITIME* as follows:

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

where, with *i=j*25*, one has:

ITIME(1+i), *ITIME(2+i)* : *clocked* CPU time in seconds and milliseconds

ITIME(3+i), *ITIME(4+i)* : *elapsed* time in seconds and milliseconds

ITIME(5+i) : number of *parallel paths* run by this processor

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

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

ITIME(8+i) : number of *waiting cycles* because this resource is being used by another processor

ITIME(9+i) ... *ITIME(22+i)* : same for critical sections resources *2* to *8*

ITIME(23+i) : information on system critical section resource

ITIME(24+i) : information on system critical section resource

ITIME(25+i) : is not used.

Most information about the algorithm performance is obtained from the *ITIME* array as it will be exemplified further on in the analysis of the algorithms.

To proceed with the actual experimentation of the Standard Explicit method, let us first consider a brief description of the implemented program which is included in the *Appendix C-IV* under the name *MB$5*[†].*STEXM*[‡]. It can be divided into *six* discrete phases from which *three* are the principal ones wherein the inherent parallelism of the implementation unfolds. For these phases the timings, speed-ups and performance analysis figures are given in the appropriate *Tables* that follow. For direct comparison reasons these phases, in all the Group Explicit algorithms that are accordingly implemented, have been set in corresponding positions and in a similar manner.

More analytically, the general *first phase* concerns the setting of the whole framework in terms of the required arrays, variables, shared data, critical sections, initialization of parallelism, the dynamically set, at each execution time, number of processors, the input data, as well as, the localization of some shared variables to diminish the overheads due to shared memory accesses.

In the *second phase* takes place the computation of the exact theoretical values at all the boundary and internal points, at the maximum time-level, using the chosen exact solution formula *(IV.B.3.1:2)*.

[†] *Directory name.*

[‡] *It stands for STandard EXplicit Method.*

The next *three* consequent phases include all the parallel work and consist of the central part of the algorithm producing the performance analysis figures. In particular, in the *third phase* takes place the computation of the exact values at all the points on both boundaries, from *zero* up to the *penultimate* time-level, using the previous exact solution formula. In the *fourth phase* takes place the computation of the exact initial values (i.e. the values at *zero* time-level) at the internal points, using again formula *(IV.B.3.1:2)*; while, in the *fifth phase* takes place the computation of the approximate values at the internal points, at every time-level, for all time-steps, using the Standard Explicit finite-difference formula *(IV.B.3.1:1)*.

The program concludes with the *sixth phase* where it takes place the output of the timing(s) and results obtained from the timed computational procedure, the computation of the maximum 'Absolute Error' - *A.E.* and the maximum 'Percentage Error' - *P.E.*, as well as the setting of the entire program's *format* statements.

Let us now restrict ourselves to the inherent parallelism of the implementation and the Standard Explicit method in specific, in terms of the most efficient utilization of system's hardware and software potential.

The explicit nature of the implementation, as well as the method's itself, offers the possibility for alternating ways of achieving the inherent parallelism. In fact, the considered each time total number of boundary and internal points can be divided into various task sizes to be assigned to the cooperating each time processors.

Basically, we have set, except the *ITIME*, three shared real arrays *U,Z,F* to be, mainly, utilized inside the timed computational procedure

of the central three parallel phases. The first array holds temporarily, at each time-level, the p.d.e.'s approximate values at the internal points, computed using the Standard Explicit finite-difference formula. The second one holds, at each time-level, the p.d.e.'s exact and approximate values at all the boundary and internal points, copied from the array F and the work-array U, respectively; while, the last array holds the p.d.e.'s exact values at the points on both boundaries, for all time-levels, computed using the chosen exact solution formula.

The p.d.e.'s exact theoretical values at all the boundary and internal points, at the maximum time-level, computed using the chosen exact solution formula, are held in a non-shared real array W; and that is because it is involved in a sequential computing procedure without any timing effect since it lies outside the timed one. For the same reason a non-shared real array *ERROR* holds the differences between the p.d.e.'s exact and approximate values at the internal points, at the maximum time-level.

After an extensive variety of experimentation we converged to the conclusion that the implementation's *granularity factor*, or otherwise the number of processes (*paths*) generated at every time-level, has to be made to be equal to the number of cooperating processors. More specifically, each path is assigned a subset of boundary or internal points equal to *ISTEP/NPROC*[†] or *NPOINT/NPROC*, respectively; without any loss of generality and mainly for a perfectly 'balanced' implementation, the total number of time-steps (*ISTEP*) and the number of internal grid points (*NPOINT*) have been chosen to be *even* and *exactly divisible* by any number of cooperating processors. Finally, to generate and

[†] *Number of PROCessors.*

terminate the *NPROC* paths each time, the *$DOPAR/$PAREND* construct is utilized, which is the most efficient and economical way to introduce parallelism in a program for the *NEPTUNE* prototype system.

- Performance Model, Experimental Results

Since our performance analysis framework will be consistently applied for all parallel algorithms implemented in the Thesis, a *complete list* of all model parameters is presented in *Table (IV.B.3.1-t1)*. To distinguish those parameters that are required by the particular algorithm in discussion each time, we shall denote them as *local* parameters, while the generally utilized ones will be denoted as *global* parameters.

In specific for the parallel algorithm for the Standard Explicit method, the experimental results obtained on the *NEPTUNE* system are presented in *Table (IV.B.3.1-t2)* along with the values of some other parameters of the performance model estimated statically.

◉ GLOBAL PARAMETERS

N_{PROCS}	:	The Number of cooperating PROCessorS in the system each run-time.
p	:	The potential of the parallel machine in terms of provided processors.
$T_c^{(e)}$	:	*Time-complexity* (experimental) of the algorithm, which is analytically distinguished to $T_s^{(e)}$ (uniprocessor time-complexity) and $T_p^{(e)}$ (*p*-processor time-complexity) (see *par.-II.B.3*).
$T_c^{(t)}$	:	*Time-complexity* (theoretical) of the algorithm, which is analytically distinguished to $T_s^{(t)}$ (uniprocessor time-complexity) and $T_p^{(t)}$ (*p*-processor time-complexity).

Table IV.B.3.1-t1: List of Parameters for the Performance Model.

C_p	:	The real *Cost* of the algorithm in relation with the number of cooperating processors each time, i.e. $C_p = p . T_p^{(e)}$.
S_p	:	*Speed-up*, the ratio between the experimental time-complexities achieved in a uniprocessor implementation and in a parallel implementation of the same algorithm; in other words, it shows the *internal acceleration* of the parallel algorithm dependent on the number of cooperating processors, i.e. $S_p = T_s^{(e)} / T_p^{(e)} (\geq 1)$.
R_{S_p}	:	*Relative* or *normalized Speed-up*, the ratio between the experimental time-complexity of the uniprocessor *standard* solution and the experimental time-complexities of the considered each time parallel algorithm achieved in a uniprocessor and parallel implementation.
$\bar{R}_{S_p}$	:	*Reference internal Speed-up*, the ratio between the experimental time-complexity of the uniprocessor *basis* solution and the experimental time-complexities achieved, e.g., in terms of r, in a uniprocessor and parallel implementation of the same algorithm.
E_p	:	*Efficiency*, the ratio of the achieved Speed-up to the corresponding number of cooperating processors, i.e. S_p/p (≤ 1); it measures the *utilization* of the parallel machine.
F_p	:	*Effectiveness*, a common basis for comparisons between various parallel algorithms for the same problem in terms of the real Cost factor, i.e. $F_p = S_p/C_p$.
N_t	:	Number (*theoretical*) of processors that is allowed by the algorithmic structure.

Table IV.B.3.1-t1(cont.d.): List of Parameters for the Performance Model.

Ac_p	:	*Algebraic-complexity* (see *par.-II.B.3*) per path of the parallel algorithm in terms of *flops* and the number of cooperating processors† in the system.
$T_p^{(t)}$	:	*Time-complexity* (theoretical) per path of the parallel algorithm in terms of the number of cooperating processors† in the system.
$N_{p(p)}$	:	Number of parallel paths allocated per processor.
$L_{O(//)}$	:	*Loops of parallelism*, i.e. the number of times the computational procedure including the *'DOPAR'* constructs is executed.
$L_{O(cs)}$	:	*Loops of critical sections*, i.e. the number of times the computational procedure including the *mutual exclusion* constructs is executed.
$R_{a(s)}$	:	*Rate of access* to the *shared* memory module in terms of *flops*.
$O_{st(s)}^{(t)}(\%)$	:	The *static (theoretical) Overhead* due to accesses to the *shared* memory module.
$R_{a(//)}$	:	*Rate of access* to the shared scheduling structure in terms of *flops* and the number of cooperating processors† in the system.
$O_{st(//)}^{(t)}(\%)$	:	The *static (theoretical) Overhead*, in terms of the number of cooperating processors† in the system, due to accesses to the shared scheduling structure.
$R_{a(cs)}$	:	*Rate of access* to *critical sections* resources in terms of *flops*.
$O_{st(cs)}^{(t)}(\%)$	:	The *static (theoretical) Overhead* due to accesses to *critical sections* resources.

Table IV.B.3.1-t1(cont.d.): List of Parameters for the Performance Model.

†*It is not always the case that the number of cooperating processors will be involved in the parametric figure.*

$S_{d(r)}$	:	Performance limitation in terms of a theoretical upper bound on the number of cooperating processors in connection with their *average excess access time* to the *Shared data resource*, on the basis that the access mechanism is independent of the number of cooperating processors.
$S'_{d(r)}$	:	Performance limitation in terms of a theoretical upper bound on the number of cooperating processors in connection with their *average access time* to the *Shared data resource*, i.e., the time to receive-serve a request, their average excess overhead due to the access mechanism being excluded.
$S_{h(r)}$	:	Performance limitation in terms of a theoretical upper bound on the number of cooperating processors in connection with the *cycle time* of the *parallel path Scheduling resource*.
m_p	:	Notation for the *maximum* theoretical number of cooperating processors due to the previous performance limitations.
f_{pt}	:	The *real time* required for a *flop* to be performed.
t_p	:	The overhead of starting and terminating a parallel path, i.e. the *cycle time* of the parallel path scheduling resource.
ℓ_{cy}	:	The *cycle time* of the *local* memory module of each processor in the system.
t_e	:	The *average excess access time* to the shared memory module for all processors in the system, compared with the average access time to their local memory modules.
s_{cy}	:	The *cycle time* of the *shared* memory module for each processor in the system.
q_a	:	Number of accesses to the shared data resource.

Table IV.B.3.1-t1(cont.d): List of Parameters for the Performance Model.

W_{st}	:	The *Wasted time* statically due to, the creation, allocation and synchronization overheads for parallel paths and, the overheads associated with accesses to the shared memory module and critical sections resources.
$n_{p(//)}$	:	Number of *parallel paths* run by each processor.
c_y	:	Number of *waiting cycles* for each processor because no parallel path is available.
t_{cy}	:	Processors *cycle time* while waiting for a parallel path to be allocated.
$Id_t^{(t)}$ (%)	:	The *average time* wasted statically (theoretical) when all but one cooperating processors are *Idle*, in terms of their relative speeds, time-complexity per parallel path, the number of parallel paths run by each processor and the loops of parallelism.
$Id_t^{(e)}$ (%)	:	The *average time* wasted dynamically (experimental) when all cooperating processors are *Idle*, in terms of the average number of waiting cycles for parallel path allocation and for access to critical sections resources.
d_l	:	Processors *idle looping time* when no parallel path is available.
t_b	:	The *blocked time* wasted by a processor to execute some shared and local instructions in the *GETRES/PUTRES* critical resource routines, when there is not any *free* block in the parallel path scheduling resource.

Table IV.B.3.1-t1(cont.d): List of Parameters for the Performance Model.

$o^{(e)}_{st(///)}(\%)$	:	The *static (experimental) Overhead* according to the number of cooperating processors in the system, due to accesses to the *parallel path* scheduling resource.
$o^{(e)}_{cn(///)}(\%)$	:	The *Overhead* occurring when processors *contend* for the 'ownership' of a *free* block in the parallel path scheduling resource.
$o^{(e)}_{tl(///)}(\%)$	:	The *total (experimental) Overhead* in the case of a uniprocessor application due to the *control* of the parallel mechanisms.
$o^{(e)}_{tl(s)}(\%)$	:	The *total (experimental) Overhead* in the case of a uniprocessor application due to the *shared data* loading in the shared memory module.
t_{cs}	:	The time to execute a *$ENTER/$EXIT*, i.e. the mutual exclusion mechanism.
n_{cs}	:	Number of *accesses* performed by each processor to *critical sections* resources.
c'_y	:	Number of *waiting cycles* for each processor to access critical sections resources.
t'_{cy}	:	Processors *cycle time* while waiting access to a critical section resource.
d'_l	:	Processors *idle looping time* when access to a critical section resource is not granted.
t'_b	:	The *blocked time* wasted by a processor to execute some similar instructions to those causing t_b in the *GETRES/PUTRES* critical resource routines, when the required critical section resource is being used by another processor.

Table IV.B.3.1-t1(cont.d.): List of Parameters for the Performance Model.

$O^{(e)}_{st(cs)}(\%)$	:	The *static (experimental) Overhead* according to the number of cooperating processors in the system, due to accesses to *critical sections* resources.
$O^{(e)}_{cn(cs)}(\%)$	:	The *Overhead* occurring when processors *contend* for the 'ownership' of a *critical section* resource.
W_{dc}	:	The *Wasted time* dynamically due to processors total idle looping time and the total time of contention for parallel paths and critical sections resources.
W	:	The total *Wasted time*, or the time the system is not used productively, as the 'sum' of the statically and dynamically wasted times.
ps	:	Processors speeds relative to the reference processor.

◉ LOCAL PARAMETERS

G_S	:	*Grid Size*, i.e. Number of Internal Points ($N_{I.P.}$) × Time-steps.
P_H	:	The *PHases* where the parallelism of the algorithm unfolds.
n_{ph}	:	The *total number* of phases where the parallelism of the algorithm unfolds.
$Ac_{(i,j)}$	:	*Algebraic-complexity* per point (or per group of two points) of the problem in terms of *flops*.
$Tc^{(t)}_{(i,j)}$	:	*Time-complexity* (theoretical) per point (or per group of two points) of the problem.
I_{cl}	:	The actual number of parallel *Implementation cycles* per section, of each phase of the algorithm, implemented in parallel, in terms of the number of cooperating processors[†] in the system.

Table IV.B.3.1-t1(cont.d.): List of Parameters for the Performance Model.

[†] *It is not always the case that the number of cooperating processors will be involved in the parametric figure.*

With respect to the Number of Internal grid Points ($N_{I.P.}$) and Time-steps[†] experimented with, the deterministic factor for the sizes of the rectangular grids considered in the *open rectangle* $[0,1]\times[0,+\infty)$ is the grid ratio r. More specifically, the grid sizes, when $r=1$, have been considered as the common basis to estimate the corresponding grid sizes for each individual method when r takes different values, since this is the maximum value for r satisfying the stability condition (i.e. $r\leq 1$) for the basic *GE* schemes upon which the more advanced *GE* schemes are dependent. Consequently, the Standard Explicit method proves in turn to be the deterministic method for the partitioning in the x-t plane each time, since it suffers from the most restrictive stability condition $r\leq\frac{1}{2}$ in order to retain reasonable accuracy. The maximum grid size allowed by the system in hand is *(1920×480)* [‡], which analogously imposes the corresponding maximum grid sizes for the *GE* methods for every value of r. Certainly, the considered numbers of internal points and time-steps could equally well be in reverse order; however, this would probably lead to less accurate solutions, and this can be checked out from the *Table* for the case of *(480×240)*. In fact, the intention of the investigation by being inclined, mainly, towards the best achievable parallel performance, in conjunction with the smallest possible time-complexity, imposed a higher priority on the number of internal points rather than the time-steps.

The combinations of the utilized processors (N_{PROCS}) have been analytically and in the given order stated, due to the occurring variations in processors relative speeds[*] (see *Table (II.B.3.1-t1)* and

---

[†] *In every program dependent performance analysis Table following, for the Number of Boundary Points we use the notation* $N_{B.P.}$.

[‡] *Internal grid Points × Time-steps.*

[*] *Processors speeds relative to the speed of the 'reference' processor* P_0 *which is normalized to be 1.000.*

Appendix C-II/par.-II.B.3.1), a fact which would certainly lead to different sets of results if other processor combinations were to be utilized.

The value for the grid ratio $r=k/h^2$, the Maximum Absolute Error obtained in each case, i.e.,

$$\max|e_{i,j}| = \max|u_{i,j}-U_{i,j}| , \qquad (IV.B.3.1:4)$$

and the corresponding Percentage Error, labelled as Maximum to indicate that it refers to the maximum absolute error occurring at this grid point, i.e.,

$$(\% \text{ Error}) = \frac{\max|e_{i,j}|}{|u_{i,j}|} \times 100 , \qquad (IV.B.3.1:5)$$

are given in the columns under the abbreviations r, $M_{A.E.}$, $M_{P.E.}$, respectively.

The concepts of the Time-complexity (Tc) and Algebraic-complexity (Ac) have been thoroughly discussed in (*par.-II.B.3*). The run-time measurements, however, obtained from the experimentation on such a parallel computer complex, as we have discussed in (*par.-II.B.3*), are not constant, since there is a variety of dynamic and unpredictable (internal and external) factors which are most likely to cause *fluctuations* in run-time. Such fluctuations are primarily due to the Operating System scheduling policies that assign certain processors to perform I/O, allocate processors to processes, etc.; in fact, even external factors, as the environmental temperature or the occasional temperature of the processors chassis, may considerably affect the run-time measurements. Therefore, this fact forced us, in all the experimental cases where a substantial discrepancy in run-times appeared, to perform a series of runs (i.e. about *4-10* repetitions), for the same case, to

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	E_p	$F_p.T_s^{(e)}$	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	E_p	$F_p.T_s^{(e)}$
240	Ø	0.5	.195053183E-01	.149965286E-03	195.383	195.383	1	1	1	.355070271E-01	.260531902E-03	385.738	385.738	1	1	1
	Ø,1				100.790	201.580	1.939	0.969	1.879			198.583	397.166	1.942	0.971	1.887
	Ø,1,2				69.720	209.160	2.802	0.934	2.618			137.003	411.009	2.816	0.939	2.642
	Ø,1,2,3				52.993	211.972	3.687	0.922	3.398			104.363	417.452	3.696	0.924	3.415
480	Ø	0.5	.106825903E-01	.106275082E-03	382.918	382.918	1	1	1	.200394355E-01	.199377537E-03	755.790	755.790	1	1	1
	Ø,1				197.950	395.900	1.934	0.967	1.871			392.053	784.106	1.928	0.964	1.858
	Ø,1,2				135.710	407.130	2.822	0.941	2.654			267.298	801.894	2.828	0.943	2.665
	Ø,1,2,3				102.730	410.920	3.727	0.932	3.473			202.205	808.820	3.738	0.934	3.493
960	Ø	0.5	.111256465E-01	.106096268E-03	759.735	759.735	1	1	1	.208908506E-01	.204622746E-03	1498.070	1498.070	1	1	1
	Ø,1				393.510	787.020	1.931	0.965	1.864			776.815	1553.630	1.928	0.964	1.860
	Ø,1,2				268.718	806.154	2.827	0.942	2.664			530.485	1591.455	2.824	0.941	2.658
	Ø,1,2,3				203.228	812.912	3.738	0.935	3.494			401.073	1604.292	3.735	0.934	3.488
1920	Ø	0.5	.115095451E-01	.103712082E-03	1513.813	1513.813	1	1	1	.227458738E-01	.204980373E-03	2987.858	2987.858	1	1	1
	Ø,1				786.073	1572.146	1.926	0.963	1.854			1548.110	3096.220	1.930	0.965	1.862
	Ø,1,2				535.260	1605.780	2.828	0.943	2.666			1054.413	3163.239	2.834	0.945	2.677
	Ø,1,2,3				404.855	1619.420	3.739	0.935	3.495			798.165	3192.660	3.743	0.936	3.503
TIME-STEPS:			120							240						

Table IV.B.3.1:-t2: Experimental Results and Performance Measurements of the Parallel Algorithm for the Standard Explicit Method on the *'NEPTUNE'* Prototype System.

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	E_p	$F_p.T_s^{(e)}$
240	Ø	0.5	.670633316E-01	.502884388E-03	766.268	766.268	1	1	1
	Ø,1				394.895	789.790	1.940	0.970	1.883
	Ø,1,2				271.700	815.100	2.820	0.940	2.651
	Ø,1,2,3				207.075	828.300	3.700	0.925	3.423
480	Ø	0.5	.364964530E-01	.363171101E-03	1500.133	1500.133	1	1	1
	Ø,1				775.268	1550.536	1.935	0.967	1.872
	Ø,1,2				532.595	1597.785	2.817	0.939	2.645
	Ø,1,2,3				402.493	1609.972	3.727	0.932	3.473
960	Ø	0.5	.407953449E-01	.399649143E-03	2974.615	2974.615	1	1	1
	Ø,1				1541.228	3082.456	1.930	0.965	1.863
	Ø,1,2				1053.690	3161.070	2.823	0.941	2.657
	Ø,1,2,3				797.498	3189.992	3.730	0.932	3.478
1920	Ø	0.5	.447628200E-01	.403463840E-03	5934.120	5934.120	1	1	1
	Ø,1				3072.303	6144.606	1.931	0.966	1.865
	Ø,1,2				2096.200	6288.600	2.831	0.944	2.671
	Ø,1,2,3				1588.353	6353.412	3.736	0.934	3.489
TIME-STEPS:			480						

Table IV.B.3.1-t2(cont.d.): Experimental Results and Performance Measurements of the Parallel Algorithm for the Standard Explicit Method on the *'NEPTUNE'* Prototype System.

produce an average run-time measurement, which would probabilistically be closer† to the real absolute value.

In respect of the *utilization* of the parallel machine in terms of the Efficiency ratio (E_p), the longer processors are *idle*, or carry out extra calculations introduced through the parallelization of the problem, the smaller it becomes.

From the Effectiveness factor point of view note that,

$$F_p = S_p/(p.T_p^{(e)}) = E_p/T_p^{(e)} = E_p.S_p/T_s^{(e)} \leq 1. \qquad (IV.B.3.1:6)$$

In other words, F_p is a measure both of Speed-up and Efficiency and consequently a parallel algorithm can accordingly be regarded as effective if it maximizes the value of this parameter. The values for the Effectiveness (multiplied by the sequential run-time) of the parallel algorithm are given in the corresponding column under the abbreviation $F_p.T_s^{(e)}$.

Some conclusions that can apparently be drawn from the examination of the previous *Table* are that, in terms of the S_p and E_p, they exhibit values very close to the optimum theoretical ones, i.e. $\leq p$ and ≤ 1, respectively, whilst $F_p.T_s^{(e)}$ also exhibits optimum results of $O(p)$. In actual fact, when increasing the number of utilized processors in each case, the real Cost of the algorithm naturally increases along with the S_p, while E_p decreases. More specifically, the obtained Speed-up values are *linear* to the number of utilized processors and quite high, producing slightly better peaks when the grid size is (*1920×240*), but only for the specified combination of *three* and *four* processors; for the combination of *two* processors the best peak achieved was for a grid of

†*Since, theoretically, only an infinite number of runs would produce the real run-time value(!).*

size (*240×240*). In consequence, for the same instances, the Efficiency and the $F_p . T_s^{(e)}$ parameters of the performance model similarly exhibit a higher peak compared to the rest; the latter factor itself indicates the grid sizes with the optimum choice of the number of processors for this computation.

With respect to the statement made previously about the trade-off occurring when choosing the maximum number of internal points instead of time-steps, it can be otherwise verified by comparing the experimentally obtained Time-complexities and real Costs for the grid of size (*480×240*) and its transposed one.

Finally, and in terms of the *convergence* of the solution of the finite-difference equation *(IV.B.3.1:1)*, the best accuracy obtained and consequently the optimum grid size was for (*1920×120*).

- Performance Analysis

In accordance with what was discussed in (*par.-II.B.3.1*) and while the parallel algorithms design has to take into account the potential parallelism, demands for shared data and demands for synchronization, it is the last factor that is the determining feature of the design of programs for such asynchronous parallel machines as the *NEPTUNE* prototype system. Thus, for all parallel systems the first two features must be taken into account, but it is only for asynchronous systems that synchronization itself is a cost. The algorithms are designed to minimize the amount of synchronization required without, however, limiting the parallelism to 2,3 and 4 processors. As it was noted in (*par.-II.B.3.1*) one function of synchronization can be to ensure that one set of processes terminates before the next set of processes is started, a fact which can result in *idle* processors.

In some instances, however, it can be possible to construct variants of the algorithms which allow, under weak conditions, the faster processors to move onto what were in the usual version the next set of processes. To ensure that each of the old processes is taken up by only one processor, *mutual exclusion* (or, otherwise, critical sections) on a list of processes is utilized, which introduces a rough correspondence between the demand for parallel paths and the demand for mutual exclusion for these algorithm versions.

With respect to the parallel algorithm for the Standard Explicit method, for a Grid of Size (*240×480*), the *Table (IV.B.3.1-t3)* being program dependent summarizes, the Algebraic-, (theoretical) Time-complexities per point and per parallel path of the algorithm, the Implementation cycles, the Number of parallel paths allocated per processor, the Loops of parallelism, the processing-to-access ratios along with their respective percentage-Overheads, the imposed performance limitations and the estimations of the Wasted time statically. While this *Table* is manually obtained from the program itself predicting somehow the experimental performance, *Table (IV.B.3.1-t4)* is system dependent and presents the 'real' performance measures obtained when 'running' the algorithm on the system. Since the estimation of most of the above figures is not simply a straightforward process and since this process will be accordingly repeated for all the parallel algorithms presented in the Thesis, we shall introduce a complete set of measuring formulae, mostly general† in nature, depending on the parameters of *Table (IV.B.3.1-t1)* and covering every possible parallel instance, even if it does not appear in this particular algorithm, for future reference.

†*It is a trivial matter to exclude from the formulae the parameters not applicable on each particular instance.*

G_S	P_H	PROCESSORS (p)		$Ac_{(i,j)}$	$Tc^{(t)}_{(i,j)}$ (secs)	I_{cl}	Ac_p	$Tc^{(t)}_p$ (secs)	$N_{p(p)}$	$I_{\alpha(//)}$	SHARED DATA		PARALLEL PATH		
		N_t	S_p								$R_{a(s)}$	$O^{(t)}_{st(s)}$	$R_{a(//)}$	$O^{(t)}_{st(//)}$	
240×480	3	$p \leq 2N_{B.P.}$ $[p \mid N_{B.P.}]$	$\emptyset(p)$	60 flops	0.023	$\frac{960}{p}$	$\frac{57.6\times10^3}{p}$ flops	$\frac{22.176}{p}$	1	1	1:60 flops	0.003%	$1:\frac{57.6\times10^3}{p}$ flops	0.005p%	
	4	$p \leq N_{I.P.}$ $[p \mid N_{I.P.}]$	$\emptyset(p)$	56 flops	0.022	$\frac{240}{p}$	$\frac{13.44\times10^3}{p}$ flops	$\frac{5.174}{p}$	1	1	1:56 flops	0.004%	$1:\frac{13.44\times10^3}{p}$ flops	0.023p%	
	5	$p \leq N_{I.P.}$ $[p \mid N_{I.P.}]$	$\emptyset(p)$	14 flops	0.005	$\frac{240}{p}$	$\frac{3.36\times10^3}{p}$ flops	$\frac{1.294}{p}$	1	480	1:2 flops	0.098%	$1:\frac{3.36\times10^3}{p}$ flops	0.093p%	

LIMITS TO PERFORMANCE			$Id^{(t)}_t$	W_{st} (secs)
$S_{d(r)}$	$S'_{d(r)}$	$S_{h(r)}$		
m_p=30,596	m_p=24,509	$m_p=\frac{18,480}{p}$	~1.5%	0.006
m_p=28,556	m_p=22,875	$m_p=\frac{4,312}{p}$		0.005
m_p=1,019	m_p=816	$m_p=\frac{1,078}{p}$		2.913

Table IV.B.3.1-t3: A Program Dependent Performance Analysis of the Parallel Algorithm for the Standard Explicit Method.

G_S	$T^{(e)}_s$ (secs)	S_p			$\sum_{i=1}^{p} c_{y_1}$	t_{cy} (μsecs)	$Id^{(e)}_t$	t_b (μsecs)	PARALLEL PATH		$T^{(e)}_s$ (secs)	PARALLEL CONTROL	$T^{(e)}_s$ (secs)	SHARED DATA	W_{dc} (secs)
	[XPFCL]	∅,1	∅,1,2	∅,1,2,3					$O^{(e)}_{st(//)}$	$O^{(e)}_{cn(//)}$	[XPFCLN]	$O^{(e)}_{tl(//)}$	[XPFCLS]	$O^{(e)}_{tl(s)}$	
240×480	766.268	1.940	2.820	3.700	5106	~10,800	~6.7%	~686	0.28%	0.42%	765.356	0.12%	764.558	0.1%	55.145

Table IV.B.3.1-t4: A System Dependent Performance Analysis of the Parallel Algorithm for the Standard Explicit Method.

For the implementation of these formulae it is essential to be given the resource provisions of the *NEPTUNE* system as were presented, but not complete[†] though, in (*Appendix C-II/par.-II.B.3.1*), while referring to the notation that can be met this is: *floating point operations - flops*, *integer operations - inops* and the ratio of *flops* to *inops* - *flips*.

However, with respect to the accuracy of the actual arithmetic of *Tables (IV.B.3.1-t3,t4)*, the experimental nature of the system along with certain existing, but not accurately known, internal limitations in Texas software (e.g. compiler, etc.) do not allow a perfect matching between the figures given in the above performance analysis *Tables*, which otherwise should theoretically coincide; they only allow an approximation that, under certain circumstances which will subsequently become apparent, proves to be very accurate.

The experimental nature of the system has been already discussed, when justifying the need for averaging the time-complexities of a set of experimental runs for an even closer approximation to the real absolute values. However, even such a procedure would not provide us with 'perfect' approximations, due to the continuous amendments, alterations and enhancements of the system's configuration inconsistently interleaving with all the research work carried out on it, which are introducing unavoidable fluctuations in most parametric figures as we shall further notice. On the other hand, it was practically impossible always to obtain their exact values at each session of experimentation, since this would require exclusive[‡] and extensive (i.e. at least 5 hours)

[†] *The complementary, but necessary, system parametric figures will be given in the text where they are accordingly required.*

[‡] *Runs on a system 'owned' exclusively by a single operator for the whole measuring session.*

runs of a plethora of specialized measurement programs; in general, the instances that this procedure has been followed become apparent from a cross-verification of the results in the corresponding performance analysis *Tables* presented each time, by applying the estimation formulae given below, using the parametric figures provided. To-date, the system with the new *ECC* memory installed exhibits those characteristics of a distributed system. Note, however, that the experimental results and performance analysis figures exhibited in the present *Chapter* were obtained prior to the installation of the *ECC* memory.

The estimation formulae for the performance analysis factors given in *Tables (IV.B.3.1-t3,t4)* will be introduced in the remainder of the paragraph, along with some particular parametric measurements essential for their proper implementation.

In respect of the *program dependent* performance analysis in *Table (IV.B.3.1-t3)*, the Algebraic-complexities are related by the formula

$$Ac_p = I_{cl} \cdot Ac_{(i,j)}, \qquad (IV.B.3.1:7)$$

which implies for the Time-complexities the formula

$$Tc_p^{(t)} = I_{cl} \cdot Tc_{(i,j)}^{(t)}$$

$$= I_{cl} \cdot Ac_{(i,j)} \cdot f_{pt} \text{ (in } \textit{microseconds}). \qquad (IV.B.3.1:8)$$

The theoretical, according to the number of cooperating processors, total Time-complexities of the algorithm, for a cross-verification with the experimental figures achieved, are given by the formula

$$Tc^{(t)} = \sum_{k=1}^{n_{ph}} Tc_{p_k}^{(t)} \cdot L_{O(//)_k}$$

$$= \sum_{k=1}^{n_{ph}} I_{cl_k} \cdot Ac_{(i,j)_k} \cdot f_{pt_k} \cdot L_{O(//)_k} < Tc^{(e)} \cdot 10^6 . \qquad (IV.B.3.1:9)$$

The *strict* inequality is mainly due to the existing, but unknown, computational bounds of the real number interpreter (i.e. the *F$RITP* subroutine) within the *FORTRAN* run-time system, as well as the ignoring of the time-complexities due to integer operations, *DO*-loop increments, local and shared transfers and the total Wasted time. Certainly a considerable time overhead occurs, caused by the call of this subroutine which has been measured by our specialized programs. However, due to optimization in the run-time system, a long sequence of *flops* is completed much more effectively than an equal number of operations timed individually. In actual fact, every time this subroutine is called the parameters of a long sequence of successive *flops* are set during this call, thus producing a series of required results at the exit (i.e. *XIT*) of the subroutine, for the *F$RWF* subroutine to write the results. Consequently, the cost of setting-up this procedure will not have to be repeated for each individual *flop*, an overhead being included in the *flops* figures given in the *Table (II.B.3.1-t1)* and in the (*Appendix C-II/par.-II.B.3.1*).

In our system measurements we succeeded to distinguish the time required for an individual *flop*, to its real absolute time and the time taken to set-up the *F$RITP* subroutine. Averaging the real absolute times for an *addition*, *subtraction* and *multiplication*, each being considered as a basic *flop* due to the close approximation between their values, we concluded to a time of ~385μs[†] for each *flop*; while the time required to set-up the subroutine, irrespective of the number of operations to be executed, was for both, the old (*parity*) memory and the newly installed *ECC* memory, ~252.6μs.

In accordance with the real absolute figure for a *division*, since

[†] *This figure has been used for our estimations in Table (IV.B.3.1-t3).*

it was quite larger than the other operations, it was converted in terms of the above *flop* figure to obtain an equivalent of ~2.5 *flops*.

However, in concern with the restrictions mentioned in the real number interpreter, we could not determine the upper bound to its total capacity† in terms of the maximum number of *flops* that it can accept at each call; what we did verify was that it is called and set-up inside any existing *DO*-loops.

In addition, when *exponential* operations are involved, this subroutine is individually called and set-up for each one of them. The measured real absolute value for each real exponentiation, after experimentation with a wide range of real numbers and given in terms of the previous *flop* figure, was equivalent to ~11.3 *flops*.

The percentage of the *static (theoretical) losses* due to accesses to the shared data resource, the shared scheduling structure and critical sections resources are, respectively, estimated by the formulae

$$o^{(t)}_{st(s)}(\%) = \frac{t_e \cdot 100}{R_{a(s)} \cdot f_{pt}} = \frac{\sum_{i=1}^{p} (s_{cy_i} - \ell_{cy_i}) \cdot 100}{p \cdot R_{a(s)} \cdot f_{pt}} , \qquad (IV.B.3.1:10)$$

$$o^{(t)}_{st(//)}(\%) = \frac{t_p \cdot 100}{R_{a(//)} \cdot f_{pt}} , \qquad (IV.B.3.1:11)$$

and

$$o^{(t)}_{st(cs)}(\%) = \frac{t_{cs} \cdot 100}{R_{a(cs)} \cdot f_{pt}} . \qquad (IV.B.3.1.12)$$

Note that, as the *Rate of access* to a shared resource we consider the *processing-to-access* ratio, i.e. the quotient of the number of *flops* over the number of accesses to this resource.

---

†*This is internal information of Texas software not available.*

With respect to the performance limitations, if the resources availability equals the total processes demand rate, then saturation occurs and no more speed-up can be achieved through utilization of more processors, despite the theoretical figure given by the N_t parameter. In an attempt to further analyze the formula presented at the end of (*par.-II.B.3.1*), we introduce some analytical formulae which correspondingly set theoretical upper bounds on the number of cooperating processors in connection with:

i) Their *average excess access time* to the shared data resource, on the basis that the access mechanism is independent of the number of cooperating processors;

ii) their *average access time* to the shared memory module, excluding their average excess overhead due to the access mechanism; and,

iii) the *cycle time* of the parallel path scheduling resource.

These formulae, respectively, are:

i) $$S_{d(r)} = \frac{R_{a(s)} \cdot f_{pt}}{t_e} = \frac{p.R_{a(s)} \cdot f_{pt}}{\sum_{i=1}^{p} (s_{cy_i} - \ell_{cy_i})} \qquad (IV.B.3.1:13)$$

ii) $$S'_{d(r)} = \frac{p.R_{a(s)} \cdot f_{pt}}{\sum_{i=1}^{p} \ell_{cy_i}} \qquad (IV.B.3.1.14)$$

iii) $$S_{h(r)} = \frac{R_{a(//)} \cdot f_{pt}}{t_p} \; . \qquad (IV.B.3.1:15)$$

For the *system dependent* performance analysis of *Table (IV.B.3.1-t4)* the *parallel path static overhead* is usually estimated in a maximum, in terms of cooperating processors, parallel implementation; according to

the information accumulated in the shared array *ITIME*, concerning the number of parallel paths run by each processor, we consider the average for all cooperating processors but P_0, since its number of paths is large compared to the others due to it includes the sequential paths in the program. Consequently, the percentage of the *parallel path static loss* is estimated by the formula

$$O^{(e)\dagger}_{st(//)}(\%) = \frac{p.\sum_{i=2}^{p} n_{p(//)_i}.t_p.100}{(p-1).\sum_{i=1}^{p} T^{(e)}_{p_i}.10^6} \quad , \qquad (IV.B.3.1{:}16)$$

where we have considered the average of the experimental timings ($T^{(e)}_p$) of all cooperating processors. For the particular case of *four* co-operating processors of *Table (IV.B.3.1-t4)* the average number of parallel paths was 483, whilst the average of the experimental timings was ~207.071 secs., these averages taken from a plethora of run-time sets. To estimate the overhead occurring when processors *contend* for the 'ownership' of a *free* block in the parallel path scheduling resource, again we consider the information accumulated in the shared array *ITIME*, which concerns the number of waiting cycles because no path is available, averaging the experimental figures of all cooperating processors, and the previous average of their experimental timings ($T^{(e)}_p$). Consequently, the percentage of the *parallel path contention loss* is given by the formula

$$O^{(e)}_{cn(//)}(\%) = \frac{\sum_{i=1}^{p} c_{y_i}.t_b.100}{\sum_{i=1}^{p} T^{(e)}_{p_i}.10^6} \quad . \qquad (IV.B.3.1{:}17)$$

---

†*A good approximation to this figure can be obtained from the* $O^{(t)}_{st(//)}$ *parameter, bearing in mind the overlapping procedure for the creation of parallel paths which we shall discuss further on.*

Note that, for the particular case of *four* cooperating processors of *Table (IV.B.3.1-t4)* the average number of wait cycles, an average figure of various experimental running sets, was ~1277. With respect to the *blocked time* t_b for the path scheduling resource, it is quite a complicated and time consuming figure to estimate[†], since it is a part of the processors whole wait cycle time (t_{cy}) along with the idle looping time (d_l) and the time it takes a *blank* routine for the basic overhead of program option servicing, which implies that all these figures have to be estimated by individual specialized procedures. In accordance with our measurements, carried out at the same period as the actual experimental work of this *Chapter*, this *blocked* time exhibited a considerable fluctuation in values around the figure of ~686μs, while the *wait cycle* as a whole around ~10800μs.

Although critical sections do not occur in our programs herein, however, for future reference we shall simply introduce the estimation formulae for their *static* and *contention* losses using again the information accumulated in the shared array *ITIME*.

For the former overhead, we consider the average of the number of accesses to critical sections resources made by each processor in a usually maximum, in terms of cooperating processors, parallel implementation.

The corresponding formula is given by

$$O^{(e)}_{st(cs)}(\%) = \frac{\sum_{i=1}^{p} n_{cs_i} . t_{cs} . 100}{\sum_{i=1}^{p} T^{(e)}_{p_i} . 10^6} \quad . \qquad (IV.B.3.1:18)$$

[†] *It is estimated indirectly depending on the figures of the other parts of the processors 'wait cycle'.*

For the *contention* loss, we consider the average of the waiting cycles for each processor due to their access to critical sections resources has not been granted, once again in a usually maximum, in terms of cooperating processors, parallel implementation. The corresponding formula is given by

$$O_{cn(cs)}^{(e)}(\%) = \frac{\sum_{i=1}^{p} c'_{y_i} . t'_b . 100}{\sum_{i=1}^{p} T_{p_i}^{(e)} . 10^6} \quad . \qquad (IV.B.3.1:19)$$

The parallel control and shared data access overheads can be estimated by using variants of the preprocessor to obtain the required information. In fact *three* load modules of the same algorithm have to be created using three different commands to compile it, the following:

(i) - '*XPFCLS*': This command generates a load module with no parallel paths involved and no shared data loading into the shared memory, which implies the run-time taken by the equivalent sequential algorithm. The rules used by this preprocessor variant to generate a sequential *FORTRAN* program from the parallel syntax are:

$DOPAR → to *DO*

$PAREND,*$JOIN* → to *CONTINUE*

$SHARED → to *COMMON/OØOX/*

$ENTER*/*$EXIT → to *CONTINUE*;

referring to the *$FORK* construct it utilizes the same type of block as the *$DOPAR* construct (see *par.-II.A.3*). It follows then that the preprocessor

can only generate a working sequential program from a parallel program that can be executed on a uniprocessor.

(ii) - *'XPFCLN'*: This command creates the same sequential load module as in *(i)*, but only the shared data will be loaded into the shared memory. Consequently, comparing the run-times achieved with this load module on a uniprocessor execution, with those of the previous command, will yield the *shared data access Overhead*. The corresponding formula is

$$O^{(e)}_{tl(s)}(\%) = \frac{(T^{(e)}_s[XPFCLN] - T^{(e)}_s[XPFCLS]).100}{T^{(e)}_s[XPFCLN]} \quad . \qquad (IV.B.3.1:20)$$

Note that, the run-times achieved through the *XPFCLN* command are certainly expected to be greater than those achieved through *XPFCLS*, but only under similar experimental conditions in accordance with what was discussed earlier herein. The results obtained can be checked from estimations of the number of shared data resource accesses made by the program, coupled with the unit (average) excess access time to shared memory; this theoretically estimated overhead is presented in *Table (IV.B.3.1-t3)*.

(iii) - *'XPFCL'*: This command, which has been discussed in (*par.-II.A.3.1*), generates a load module as in *(ii)*, with the additional overhead of parallel paths creation,

allocation and termination. Consequently, if we compare the run-times achieved using the *XPFCLN* command with those achieved through the *XPFCL* command, the time discrepancy will be the *static parallel control access Overhead*. The corresponding formula is

$$O^{(e)}_{tl(//)}(\%) = \frac{(T^{(e)}_s[XPFCL] - T^{(e)}_s[XPFCLN]).100}{T^{(e)}_s[XPFCL]^{\dagger}} \quad .$$

(IV.B.3.1:21)

Again, the longest run-times are naturally expected to be those achieved through the *XPFCL* command, but only under similar experimental conditions. Once again, the results obtained can be checked from estimations of the number of parallel path scheduling resource accesses and the critical section 'entries/exits' made by the program, coupled with their measured unit time cost; this theoretically estimated overhead is presented in *Table (IV.B.3.1-t3)*.

Let us now present a general formula which will express the entire performance of a Multiprocessor computer complex in terms of the time not used productively, i.e. the Wasted time (W) in total; this formula is

$$W = W_{st} + W_{dc} = \sum_{i=1}^{p} T^{(e)}_{p_i} - T^{(e)}_s \quad , \qquad \textit{(IV.B.3.1:22)}$$

---

†*Note that, the 'XPFCL' time given in Table (IV.B.3.1-t4) represents the time of the slowest processor in the system.*

which is the *sum* of times taken by the p processors to complete their subtasks, less the uniprocessor time. The algorithm design time, certainly, has been ignored.

The *Wasted time statically* (W_{st}), due to the creation, allocation and synchronization overheads for parallel paths and the overheads associated with accesses to the shared memory and critical sections resources, can be estimated in a program and system resource provisions dependent manner.

The *Wasted time dynamically* (W_{dc}) refers to the time in total that all cooperating processors are idle 'waiting' for subtasks to be allocated and for access to the critical sections resources; or rather more specific, to the time they waste to perform idle loops and contend for the 'ownership' of shared resources.

With respect to the former *Wasted time statically* two estimation formulae can be given cross-verifying each other:

$$i)\quad W_{st} = p.(q_a.I_{cl}.N_{p(p)}.L_{O(//)}.t_e + t_p.N_{p(p)}.L_{O(//)} + t_{cs}.N_{p(p)}.L_{O(cs)})$$

$$= N_{p(p)} \cdot [q_a.I_{cl}.L_{O(//)} \cdot \sum_{i=1}^{p} (s_{cy_i} - \ell_{cy_i}) + p.(t_p.L_{O(//)} + t_{cs} \cdot L_{O(cs)})] \ , \qquad (IV.B.3.1:23)$$

and, in particular for the parallel algorithms of this *Chapter*,

$$ii)\quad W_{st} = \frac{p.Tc_p^{(t)}.N_{p(p)}.L_{O(//)}.(O_{st(s)}^{(t)} + O_{st(//)}^{(t)})}{100}$$

$$= \frac{p.I_{cl}.Tc_{(i,j)}^{(t)}.N_{p(p)}.L_{O(//)}.(O_{st(s)}^{(t)} + O_{st(//)}^{(t)})}{100}$$

$$= \frac{I_{cl}.Ac_{(i,j)}.N_{p(p)}.L_{O(//)}.(\sum_{i=1}^{p} (s_{cy_i} - \ell_{cy_i}).R_{a(//)} + p.t_p.R_{a(s)})}{R_{a(s)}.R_{a(//)}} \ ,$$

$$(IV.B.3.1:24)$$

where the time figures for both formulae are expressed in microseconds (μs). Note that in *Table (IV.B.3.1-t3)* this parameter (in *secs*) is given for the case of all *four* processors cooperating. In specific with formula *i)* as $q_a^{\dagger}$ we should not consider the *normalized* rate of access to the shared data resource. In addition, we have considered that parallel paths are created individually one after the other in succession, which basically is not true since there is an overlapping procedure (not accurately estimatable though) which reduces the total parallel mechanism overhead to strictly $<p.t_p$ $(p>1)$.

For the *average Idle time* $(Id_t^{(t)})$ of all but one cooperating processors, a relevant estimation figure, complementary to the figure for the *Wasted time statically* (W_{st}), can be obtained. More specifically, if p subtasks are not available at any time, in a p-processor system, then all processors cannot be processing and thus some of them must be *idle*. All processors can only be processing at every stage if subtasks can be allocated to them, taking into account subtasks length and processors relative speed for the particular set of data and the specific hardware involved (see *par.-II.B.3.1*). Consequently, for different processors of different relative speeds, the *faster* processor will finish processing before the *slower* one and so it will be forced to wait for the slower processor. This is apparent for the synchronous algorithms implemented on *MIMD* complexes of processors.

Therefore it follows that if the relative speeds of processors are known, a *static* estimation of the *average Idle time* of all but one cooperating processors can be given (rather predicted) by examining

---

† *For this particular parallel algorithm the actual processing-to-access ratio is 14 flops over 7 accesses to the shared data resource, for each implementation cycle.*

the time-complexity per parallel path, the number of parallel paths run by each processor and the loops of parallelism. However, this method provides us with only a fair approximation to the actual *average Idle time* of all but one cooperating processors; this is due to all the factors discussed previously, as well as others ignored which contribute to increase the run-time interval between fastest-slowest processor, such as, the time-complexities due to integer operations, *DO*-loop increments, local and shared transfers, etc.

The formula for the *static* estimation of the *average Idle time* ($Id_t^{(t)}$) of all but one cooperating processors, in specific for the algorithms of this *Chapter*, is

$$Id_t^{(t)}\ (\%) = \frac{(maxps\text{-}minps)\cdot \sum_{k=1}^{n_{ph}} Tc_{p_k}\cdot N_{p(p)_k}\cdot L_{O(//)_k}\cdot 100}{T_p^{(e)}}$$

$$= \frac{(maxps\text{-}minps)\cdot \sum_{k=1}^{n_{ph}} Icl_k\cdot Ac_{(i,j)_k}\cdot f_{pt_k}\cdot N_{p(p)_k}\cdot L_{O(//)_k}\cdot 100}{T_p^{(e)}\cdot 10^6},$$

(IV.B.3.1:25)

where as $T_p^{(e)}$ has been considered the smallest run-time with p-processors. For the figure given in *Table (IV.B.3.1-t3)* we have considered the case that all *four* processors of the *NEPTUNE* system were cooperating, while $T_p^{(e)}$, an average of the smallest run-time figures of a plethora of running sets with *four* processors, was 207.065 secs.

On the other hand, in respect of the *total Idle time* (i.e. idle looping time+blocked time) for the estimation of the *Wasted time dynamically* (W_{dc}), this parameter can be easily estimated with a *system dependent* method since its character is completely experimental depending

upon the outcome of running the algorithm on the system. According to this method, the shared array *ITIME* is scanned once more for the information concerning the number of *waiting cycles* for each processor, due to the fact that no parallel path was available and no access to critical sections resources was granted. Consequently, the corresponding formula for the *Wasted time dynamically* (W_{dc}) is

$$W_{dc} = \sum_{i=1}^{p} (c_{y_i} . t_{cy_i} + c'_{y_i} . t'_{cy_i})$$

$$= \sum_{i=1}^{p} [c_{y_i} . (d_{l_i} + t_{b_i}) + c'_{y_i} . (d'_{l_i} + t'_{b_i})] , \qquad (IV.B.3.1:26)$$

which, due to the lack of critical sections resources in the algorithms of this *Chapter*, simplifies as

$$W_{dc} = \sum_{i=1}^{p} c_{y_i} . (d_{l_i} + t_{b_i})^{\dagger} \text{ (in } microseconds) . \qquad (IV.B.3.1:27)$$

Note that, for the figure presented in *Table (IV.B.3.1-t4)* as the *total* number of wait cycles (i.e. 5106) we considered the average figure of various experimental running sets with all *four* processors of the *NEPTUNE* system cooperating.

Finally, an alternate indicative parameter of the *average Idle time* ($Id_t^{(e)}$) of all the cooperating processors in the system can be *dynamically* estimated considering their average number of waiting cycles for parallel path allocation and for access to critical sections resources from the shared array *ITIME*, coupled with their respective wait cycle time-unit, i.e.,

$$Id_t^{(e)}(\%) = \frac{\sum_{i=1}^{p} (c_{y_i} . t_{cy_i} + c'_{y_i} . t'_{cy_i}) . 100}{\sum_{i=1}^{p} T_{p_i}^{(e)} . 10^6} . \qquad (IV.B.3.1:28)$$

---

†*The basic time overhead of program option servicing (i.e. 'blank' routine) does not appear, but it is included in the actually used figure of processors 'cycle time'.*

For the figure given in *Table (IV.B.3.1-t4)* we have considered the case that all *four* processors of the *NEPTUNE* system were cooperating, while, once more, we have taken the average of their experimental timings ($T_p^{(e)}$) given earlier herein. This parameter† assists in a very good approximation to the total *Wasted time dynamically* (W_{dc}), which can be verified from the latter figure in the same *Table*. Also, the *sum* of the wasted times statically and dynamically gives us a quite good approximation to the total Wasted time (W) estimated through the formula *(IV.B.3.1:22)*. In fact, the figure found according to this formula, using the above average of experimental timings ($T_p^{(e)}$) of all co-operating processors, was ~62.016 secs.

In conclusion, for the verification of each of the figures given in *Tables (IV.B.3.1-t3,t4)*, the interested reader should follow each procedure, leading to the specific parameter, in an analytical manner, from the start, to obtain the accurate values for the interleaving factors in the formulae, since the round-off errors in computations are bound to introduce slight discrepancies when substituting the factors as they appear in the *Tables*.

IV.B.3.2: The 'Group Explicit With Ungrouped Ends' - *GEU* Method: Experimental Results And Performance Analysis On The '*NEPTUNE*' Prototype System

As previously mentioned in (*par.-IV.B.3*), in the case that the line segment $0 \leq x \leq 1$ is divided into an *odd* number m of equal sub-intervals, the concept of the *GE* class of methods can be similarly implemented and in fact the resulting schemes are much more balanced and computationally

† *When the corresponding timing is multiplied by the number of cooperating processors in the system.*

preferable for *MIMD* complexes of processors. Hence, at every time-level, the number of internal points is even, i.e. *(m-1)*. This gives, at every time-level, either *(m-1)/2* complete groups of two points, or *(m-3)/2* groups of two points and one ungrouped point adjacent to each boundary.

The first *GE* scheme for this particular case, examined herein, is obtained by using, at every time-level, either of equations *(IV.B.3:11,21)* at the *left* ungrouped point, *(m-3)/2* times either of the systems of equations *(IV.B.3:17,18)* from the second internal point to the $(m-2)^{th}$ point and either of equations *(IV.B.3:9,20)* at the last $(m-1)^{th}$ point which is left ungrouped at the far *right* of the line, adjacent to the boundary. This scheme in implicit matrix form, for the particular case of Burgers' equation *(IV.B.2:2)*, similarly as for the *even* number of equal sub-intervals case, is given by

$$(I+r\hat{G}_{1,j})\underline{U}_{j+1} = (I-r\hat{G}_{2,j})\underline{U}_j + \underline{b}_3 \qquad \textit{(IV.B.3.2:1)}$$

where

$$\underline{b}_3^T = [\hat{b}_2^{(n)} rU_{0,j+1}, 0, \ldots, 0, \hat{a}_1^{(n)} rU_{m,j+1}] \ , \qquad \textit{(IV.B.3.2:2)}$$

and

$$\hat{G}_{1,j} = \begin{bmatrix} \hat{b}_2^{(n)} & & & & \\ & G^{(1)} & & & \\ & & G^{(2)} & & \\ & & & \ddots & \\ & & & & G^{\frac{1}{2}(m-3)-1} \\ & & & & G^{\frac{1}{2}(m-3)} \\ & & & & & \hat{a}_1^{(n)} \end{bmatrix} \qquad \textit{(IV.B.3.2:3)}$$

$$\hat{G}_{2,j} = \begin{bmatrix} \dot{G}^{(1)} & & & \\ & \dot{G}^{(2)} & & \\ & & \ddots & \\ & & & \dot{G}^{\frac{1}{2}(m-2)} \\ & & & & \dot{G}^{\frac{1}{2}(m-1)} \end{bmatrix} \ , \qquad \textit{(IV.B.3.2:4)}$$

while $G^{(k)}$, $k=1,2,\ldots,\frac{1}{2}(m-3)$, $\mathring{G}^{(l)}$, $l=1,2,\ldots,\frac{1}{2}(m-1)$, for $j=0,1,2,\ldots$, are the (2×2) matrices defined in (*par.-IV.B.3*). This scheme is described by the *brick* diagram in *Figure (IV.B.3.2-f1)*.

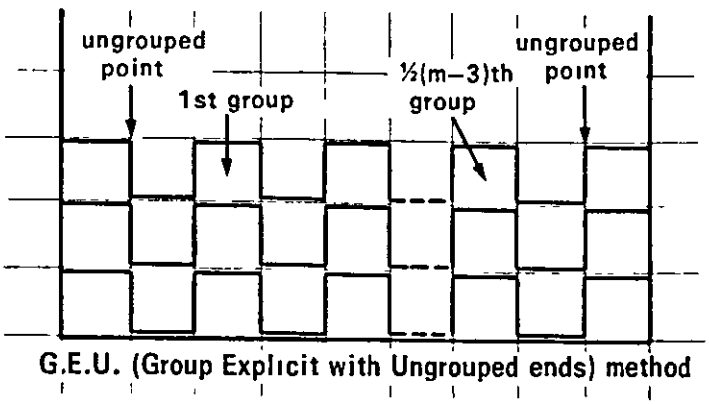


Figure IV.B.3.2-f1: The Representative Diagram of this Scheme.

From the aspect of parallel programming of all the resulting *GE* schemes, under the previous assumption for an *even* number of internal points and for the particular case of Burgers' equation, formulae *(IV.B.3:18,20,21)*† have been implemented in an unavoidable, but highly efficient, *synchronized* manner which is due to the nature of the methods.

To proceed with the actual experimentation of the *GEU* method, the implemented program of which is included in the *Appendix C-IV* under the name *MB$5.GEUONI*‡ and in accordance with what we have mentioned for the *MB$5.STEXM* program, it can be divided into *six* corresponding discrete phases from which, again, *three* are the principal ones wherein the inherent parallelism of the implementation unfolds.

The first two phases are similar with those of the *MB$5.STEXM*

†*Not always required all of them.*

‡*It stands for Group Explicit with Ungrouped ends method for Odd Number of Intervals.*

program except that, in the *second phase*, we do not compute the exact theoretical values at the boundary points, at the maximum time-level.

To proceed with the phases which include all the parallel work and comprise the central part of the algorithm producing the performance analysis figures, we have, in the *third phase*, the computation of the exact values at all the points on both boundaries, for all time-levels, starting from time-level *one* up to the maximum time-level, using, as in the previous phase, the chosen exact solution formula *(IV.B.3.1:2)*. In the *fourth phase* an identical computation to the corresponding phase of *MB$5.STEXM* program occurs, while in the *fifth phase* takes place the computation of the approximate values at the internal points, at every time-level, for all time-steps, using the Group Explicit finite-difference and Saul'yev's asymmetric formulae, accordingly. Note that in accordance with what was discussed in *(par.-IV.B.3)*, the values of $\hat{a}_1, \hat{a}_2, \hat{b}_1, \hat{b}_2$ have been computed using the formulae given in *(IV.B.3:16)*.

The program is brought to a similar conclusion in the last phase by outputting the timing(s) and results obtained from the timed computational procedure or computed there in this phase, i.e. the maximum *A.E.* and the maximum *P.E.*, as well as setting the entire program's *format* statements.

From the aspect of the inherent parallelism of the implementation and in terms of the most efficient utilization of system's hardware and software potential, the explicit nature of the implementation, as well as the method's itself, offers a plethora of alternating ways to exploit the parallelism concerning the task sizes that the total number of boundary and internal points can be divided into, to be allocated to the cooperating processors. After an extensive range of experiments

we reached the conclusion that the implementation's *granularity factor* has to be made to be equal to the number of cooperating processors, in terms of the subset sizes of the boundary and internal points. Again, without any loss of generality and for a 'balanced' implementation, the considered total number of time-steps (*ISTEP*) and the number of internal grid points (*NPOINT*) have been chosen to be *even* and *exactly divisible* by any number of cooperating processors. To generate and terminate the *NPROC* paths each time, the *$DOPAR/$PAREND* construct is utilized, which is the most efficient and economical way to introduce parallelism in a program for the *NEPTUNE* prototype system.

Finally, in concern with the shared arrays utilized, again, except for the *ITIME* array, three shared real arrays *U,Z,F* have been declared for similar purposes; on the other hand, two non-shared real arrays *W,ERROR* are utilized which being involved in a sequential computing procedure respectively hold, the p.d.e.'s exact theoretical values at the internal points only, at the maximum time-level, computed using the exact solution formula, and the differences between the p.d.e.'s exact and approximate values at the internal points, again at the maximum time-level.

- Experimental Results

The experimental results obtained from the parallel algorithm for the *GEU* method are presented in *Table (IV.B.3.2-t1)* along with the values of other parameters of the performance model estimated statically.

The concepts of the parameters appearing in this *Table* are as they were presented in (*par.-IV.B.3.1*), with the exception of a new introduced parameter R_{S_p} which mirrors the *Relative* or *normalized Speed-up*,

obtained comparing the experimental time-complexity of the uniprocessor solution using the Standard Explicit method, to the experimental time-complexities of the present parallel algorithm achieved in a uniprocessor and parallel implementation.

The experimented grid sizes, when the grid ratio r=1, have been accordingly set to correspond to the grid sizes chosen for the Standard Explicit method, where $r=\frac{1}{2}$, for a direct comparison.

Some conclusions that can be apparently drawn from the examination of *Table (IV.B.3.2-t1)* are that, in terms of the S_p and E_p, they exhibit values very close to the optimum theoretical ones, i.e. $\leq p$ and ≤ 1, respectively, whilst $F_p.T_s^{(e)}$ also exhibits optimum results of $O(p)$.

In comparison with the parallel algorithm for the Standard Explicit method (hereafter being called the *standard* algorithm) all these parameters produce considerably better figures for the present algorithm; however, the extraordinary thing to be noted is that these better results have been obtained with smaller running time-complexities, for every number of cooperating processors, a fact which agrees with the implementor's should be real goal to minimize execution time and not simply to maximize Speed-up. In fact, the *Relative Speed-up* (R_{S_p}) parameter very clearly shows that the $O(p)$ speed-up limit has been well surpassed and is being established to $>O(r.p)$ as it will become even clearer in the subsequent algorithms.

The same observation, with that for the *standard* algorithm, can be made concerning the real Cost (C_p) of the present algorithm which naturally increases, when increasing the number of cooperating processors, along with the S_p, while E_p decreases. In particular with the obtained Speed-up values, which are *linear* to the number of cooperating processors

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	E_p	$F_p.T_s^{(e)}$	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	E_p	$F_p.T_s^{(e)}$
240	Ø	1.0	.15202382E-01	.11032819E-03	182.435	182.435	1	1.071	1	1	.30064404E-01	.22059679E-03	361.038	361.038	1	1.068	1	1
	Ø,1				93.405	186.810	1.953	2.092	0.977	1.907			184.615	369.230	1.956	2.089	0.978	1.912
	Ø,1,2				63.180	189.540	2.888	3.092	0.963	2.779			124.425	373.275	2.902	3.100	0.967	2.807
	Ø,1,2,3				47.935	191.740	3.806	4.076	0.951	3.621			94.075	376.300	3.838	4.100	0.959	3.682
480	Ø	1.0	.59258490E-02	.59247016E-04	363.390	363.390	1	1.054	1	1	.10997057E-01	.10997057E-03	716.290	716.290	1	1.055	1	1
	Ø,1				186.390	372.780	1.950	2.054	0.975	1.901			367.170	734.340	1.951	2.058	0.975	1.903
	Ø,1,2				124.660	373.980	2.915	3.072	0.972	2.833			246.360	739.080	2.907	3.068	0.969	2.818
	Ø,1,2,3				93.990	375.960	3.866	4.074	0.967	3.737			185.750	743.000	3.856	4.069	0.964	3.718
960	Ø	1.0	.65891742 7E-02	.65743923 2E-04	724.058	724.058	1	1.049	1	1	.13098143 0E-01	.13071298 6E-03	1424.425	1424.425	1	1.052	1	1
	Ø,1				369.803	739.606	1.958	2.054	0.979	1.917			725.733	1451.466	1.963	2.064	0.981	1.926
	Ø,1,2				248.258	744.774	2.917	3.060	0.972	2.835			488.213	1464.639	2.918	3.068	0.973	2.838
	Ø,1,2,3				186.598	746.392	3.880	4.072	0.970	3.764			368.608	1474.432	3.864	4.064	0.966	3.733
1920	Ø	1.0	.69013424 2E-02	.68962574 0E-04	1449.140	1449.140	1	1.045	1	1	.12705136 1E-01	.12695789 3E-03	2856.785	2856.785	1	1.046	1	1
	Ø,1				735.848	1471.696	1.969	2.057	0.985	1.939			1453.493	2906.986	1.965	2.056	0.983	1.932
	Ø,1,2				495.085	1485.255	2.927	3.058	0.976	2.856			974.453	2923.359	2.932	3.066	0.977	2.865
	Ø,1,2,3				372.115	1488.460	3.894	4.068	0.974	3.791			733.708	2934.832	3.894	4.072	0.973	3.790
TIME-STEPS:			60								120							

Table IV.B.3.2-t1: Experimental Results and Performance Measurements of the Parallel Algorithm for the *G.E.U.* Method on the *'NEPTUNE'* Prototype System.

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$T_c^{(e)}$ (secs)	C_p	S_p	R_{S_p}	E_p	$F_p.T_s^{(e)}$
240	Ø	1.0	.568413064E-01	.426232815E-03	715.018	715.018	1	1.072	1	1
	Ø,1				367.410	734.820	1.946	2.086	0.973	1.894
	Ø,1,2				248.180	744.540	2.881	3.088	0.960	2.767
	Ø,1,2,3				187.918	751.672	3.805	4.078	0.951	3.619
480	Ø	1.0	.231868960E-01	.230729580E-03	1416.660	1416.660	1	1.059	1	1
	Ø,1				726.310	1452.620	1.950	2.065	0.975	1.902
	Ø,1,2				487.400	1462.200	2.907	3.078	0.969	2.816
	Ø,1,2,3				368.120	1472.480	3.848	4.075	0.962	3.702
960	Ø	1.0	.261719599E-01	.261187553E-03	2827.950	2827.950	1	1.052	1	1
	Ø,1				1448.135	2896.270	1.953	2.054	0.976	1.907
	Ø,1,2				969.343	2908.029	2.917	3.069	0.972	2.837
	Ø,1,2,3				730.743	2922.972	3.870	4.071	0.967	3.744
1920	Ø	1.0	.242411233E-01	.242233276E-03	5667.958	5667.958	1	1.047	1	1
	Ø,1				2876.640	5753.280	1.970	2.063	0.985	1.941
	Ø,1,2				1932.828	5798.484	2.932	3.070	0.977	2.866
	Ø,1,2,3				1454.243	5816.972	3.898	4.081	0.974	3.798
TIME-STEPS:			240							

Table IV.B.3.2-t1(cont.d.): Experimental Results and Performance Measurements of the Parallel Algorithm for the *G.E.U.* Method on the *'NEPTUNE'* Prototype System.

each time, they produce slightly better peaks when the grid size is *(1920×240)*, for every specified combination of processors, while, as the number of internal points increases, the Speed-ups achieved with the stated combination of 3 and 4 cooperating processors analogously and smoothly increase, compared with the fluctuating figures of the *standard* algorithm. In consequence, for the same instances, the Efficiency and the $F_p.T_s^{(e)}$ parameters of the performance model similarly exhibit a continuous increase. The latter factor examined alone indicates the optimum, allowed by the system, grid size in terms of the acceleration in computation.

Finally, and in terms of the *convergence* of the solution achieved through the *GEU* method, the best accuracy obtained and consequently the optimum grid size was for *(480×60)*.

- Performance Analysis

In accordance with the list of parameters for the performance model presented in *Table (IV.B.3.1-t1)* and the corresponding measuring formulae introduced in *(par.-IV.B.3.1)*, the program and system dependent performance analyses figures are given in *Tables (IV.B.3.2-t2,t3)*[†], respectively.

The complementary information to that given in *(Appendix C-II/par.-II.B.3.1)* is as it was presented in *(par.-IV.B.3.1)*. In particular for the information obtained from the shared array *ITIME* when testing the parallel algorithm for the *GEU* method, a plethora of runs was carried out always considering the average[‡] figures accordingly. This information being vital for the estimation of the experiment dependent performance

---

[†] *Note that, in the case of the parameters* $Ac_{(i,j)}$, $Tc_{(i,j)}^{(t)}$, *for the fifth phase of the algorithm, we have considered the corresponding complexities per group of two internal points, at each time-level, the rest of the parameters computed accordingly.*

[‡] *This also concerns the timings presented in Table (IV.B.3.2-t3) obtained from running the load modules generated through the 'XPFCL','XPFCLN' and 'XPFCLS' commands.*

G_S	P_H	PROCESSORS (p)		$Ac_{(i,j)}$	$Tc^{(t)}_{(i,j)}$ (secs)	I_{cl}	Ac_p	$Tc^{(t)}_p$ (secs)	$N_{p(p)}$	$T_{cc(//)}$	SHARED DATA		PARALLEL PATH		
		N_t	S_p								$R_{a(s)}$	$O^{(t)}_{st(s)}$	$R_{a(//)}$	$O^{(t)}_{st(//)}$	
240×240	3	$p \leq 2N_{B.P.}$ $[p \mid N_{B.P.}]$	$O(p)$	60 flops	0.023	$\frac{480}{p}$	$\frac{28.8\times10^3}{p}$ flops	$\frac{11.088}{p}$	1	1	1:60 flops	0.003%	1:$\frac{28.8\times10^3}{p}$ flops	0.011p%	•••
	4	$p \leq N_{I.P.}$ $[p \mid N_{I.P.}]$	$O(p)$	56 flops	0.022	$\frac{240}{p}$	$\frac{13.44\times10^3}{p}$ flops	$\frac{5.174}{p}$	1	1	1:56 flops	0.004%	1:$\frac{13.44\times10^3}{p}$ flops	0.023p%	
	5	$p \leq N_{I.P.}$ $[p \mid N_{I.P.}]$	$O(p)$	49 flops	0.019	$\frac{120}{p}$	$\frac{5.88\times10^3}{p}$ flops	$\frac{2.264}{p}$	1	240	1:4 flops	0.049%	1:$\frac{5.88\times10^3}{p}$ flops	0.053p%	

Table IV.B.3.2-t2: A Program Dependent Performance Analysis of the Parallel Algorithm for the *G.E.U.* Method.

LIMITS TO PERFORMANCE			$I_{d^{(t)}_t}$	W_{st} (secs)
$S_{d(r)}$	$S'_{d(r)}$	$S_{h(r)}$		
m_p=30,596	m_p=24,509	$m_p=\frac{9,240}{p}$	~1.4%	0.005
m_p=28,556	m_p=22,875	$m_p=\frac{4,312}{p}$		0.005
m_p=2,039	m_p=1,633	$m_p=\frac{1,886}{p}$		1.413

G_S	$T^{(e)}_s$ (secs)	S_p			$\sum_{1=1}^{p} c_{y_1}$	t_{cy} (μsecs)	$Id^{(e)}_t$	t_b (μsecs)	PARALLEL PATH		$T^{(e)}_s$ (secs)	PARALLEL CONTROL	$T^{(e)}_s$ (secs)	SHARED DATA	W_{dc} (secs)
	[XPFCL]	ø,1	ø,1,2	ø,1,2,3					$O^{(e)}_{st(//)}$	$O^{(e)}_{cn(//)}$	[XPFCLN]	$O^{(e)}_{tl(//)}$	[XPFCLS]	$O^{(e)}_{tl(s)}$	
240×240	715.018	1.946	2.881	3.805	2712	~10,800	~3.9%	~686	0.16%	0.25%	714.163	0.12%	713.693	0.07%	29.290

Table IV.B.3.2-t3: A System Dependent Performance Analysis of the Parallel Algorithm for the *G.E.U.* Method.

analysis parameters, for the case that all *four* processors of the *NEPTUNE* system were cooperating was as follows:

i) The smallest run-time $T_p^{(e)}$ to be utilized in formula *(IV.B.3.1:25)* was ~187.908 secs;

ii) the total number of wait cycles to be utilized in formula *(IV.B.3.1:27)* was ~2712, which implied an average number of wait cycles per processor of ~678;

iii) the average experimental timing of all cooperating processors was ~187.912 secs; and,

iv) the number of parallel paths run by each processor, considering the average of all cooperating processors but P_0, was 243.

Finally, the time the system was not used productively (W) being estimated through the formula *(IV.B.3.1:22)*, by using the average experimental timing in *iii)*, was~36.630 secs; a good approximation to this total wasted time can be obtained from the *sum* of the wasted times statically and dynamically given in the performance analysis *Tables*.

To conclude, with respect to the *Table (IV.B.3.2-t2)*, note that the normalization and integer rounding of the number of accesses to the shared data resource, in the *fifth phase* of the algorithm, introduce slight discrepancies between the results found through the cross-verifying formulae *(IV.B.3.1:23,24)* for the wasted time statically. However, if the real processing-to-access ratio is used, i.e. 49 *flops* over 12 accesses to the shared data resource, for each implementation cycle, then both results coincide to the 1.413 secs figure.

IV.B.3.3: THE 'GROUP EXPLICIT COMPLETE' - *GEC* METHOD: EXPERIMENTAL RESULTS AND PERFORMANCE ANALYSIS ON THE *'NEPTUNE'* PROTOTYPE SYSTEM

This scheme is obtained by using $(m-1)/2$ times either of the systems of equations *(IV.B.3:17,18)* for the first to $(m-1)^{th}$ point, at every time-level, and in implicit matrix form, for the particular case of Burgers' equation and in accordance with the previous definition of matrices $\hat{G}_{1,j}, \hat{G}_{2,j}$, is given by

$$(I+r\hat{G}_{2,j})\underline{U}_{j+1} = (I-r\hat{G}_{1,j})\underline{U}_j + \underline{b}_4 \qquad \textit{(IV.B.3.3:1)}$$

where

$$\underline{b}_4^T = [\hat{b}_2^{(n)} rU_{0,j}, 0, \ldots, 0, \hat{a}_1^{(n)} rU_{m,j}] \ . \qquad \textit{(IV.B.3.3:2)}$$

This scheme is described by the *brick* diagram in *Figure (IV.B.3.3-f1)*.

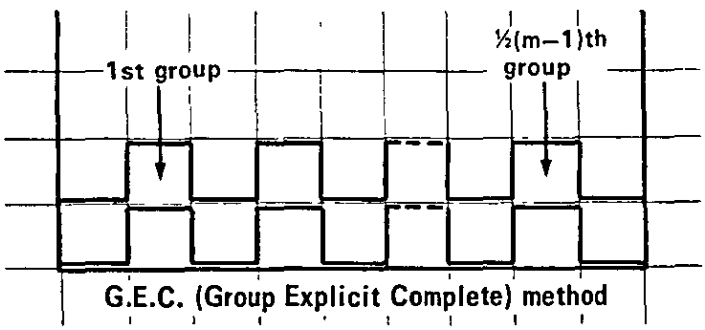


Figure IV.B.3.3-f1: The Representative Diagram of this Scheme.

To proceed with the actual implementation of the *GEC* method, the implemented program, which is included in *Appendix C-IV* under the name *MB$5.GECONI*†, has a similar phase-structure to that of the previous parallel algorithms in this *Chapter*.

In a general comparison of this implementation with the one for the *GEU* method and in outline terms of the differences occurring, in the

†*It stands for Group Explicit Complete method for Odd Number of Intervals.*

second phase, we do compute the exact theoretical values at all the boundary and internal points, at the maximum time-level, using the chosen exact solution formula *(IV.B.3.1:2)*. In the *third phase*, the computation of the exact values at all the points on both boundaries starts from time-level *zero* up to the penultimate time-level using again the above formula. In particular for the *fifth phase*, where the main part of the inherent parallelism lies, this algorithm requires less programming effort than that for the *GEU* method, since there are no ungrouped points and only the Group Explicit finite-difference formulae in *(IV.B.3:18)* are used at each time-level. Note, again, that the values of $\hat{a}_1, \hat{a}_2, \hat{b}_1, \hat{b}_2$ have been computed using the formulae given in *(IV.B.3:16)*.

Once again, and similar for all the remaining *GE* alternating schemes introduced in this *Chapter*, an extensive range of experiments led us to the same conclusion about the implementations' *granularity factor* as for the previous two parallel algorithms. The same justification therein about the choice of the number of time-steps (*ISTEP*) and the number of internal grid points (*NPOINT*) for a *balanced* implementation, due to the natural behaviour of *MIMD* systems, will also stand for all algorithms herein. In addition, to obtain always the same running effects, for fair comparison purposes, the *$DOPAR/$PAREND* construct is throughout utilized as before.

Finally, in concern with the declared shared and non-shared arrays in the program, the only difference, compared to the *GEU* method's programming, occurs in the real non-shared array *W* which will hold the p.d.e.'s exact theoretical values at all the boundary and internal points, at the maximum time-level, computed using formula *(IV.B.3.1:2)*.

- Experimental Results

The experimental results obtained from the parallel algorithm for the *GEC* method are presented in *Table (IV.B.3.3-t1)* along with the values of other parameters of the performance model estimated statically, their concepts being those introduced for the previous parallel algorithms. The experimented grid sizes have been similarly set according to the grid ratio r=1, for a direct comparison of the achieved performance analysis results with those of the other implemented methods herein.

The S_p and E_p parameters exhibit values very close to the optimum theoretical ones, i.e. $\leq p$ and ≤ 1, respectively, along with $O(p)$ results for the $F_p.T_s^{(e)}$ factor.

In comparison with the *standard* parallel algorithm, all these parameters produce much better figures for the present algorithm, with, again, considerably smaller running time-complexities achieved for all numbers of cooperating processors. This observation receives even more significance if these running time-complexities are compared with the corresponding ones achieved from the parallel algorithm for the *GEU* method. In actual fact, the time-complexities of the present algorithm are also considerably smaller than those achieved running the latter one, despite the fact that the S_p and E_p exhibit generally lower optimum values. The programming simplicity of the algorithm for the *GEC* method, compared to that for the *GEU* method, has resulted in smaller timings, while, on the contrary, the comparatively excessive coding in the latter's implementation resulted in a greater seemingly internal acceleration. However, once again, the implementor's should be real goal has been truly achieved and this is exhibited from the even higher values, for the present algorithm, of the *Relative Speed-up* (R_{S_p}) parameter their range being similarly $>O(r.p)$.

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	E_p	$F_p.T_s^{(e)}$	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	E_p	$F_p.T_s^{(e)}$
240	Ø	1.0	.917399675E-02	.665783882E-04	167.940	167.940	1	1.163	1	1	.183343291E-01	.134527683E-03	330.368	330.368	1	1.168	1	1
	Ø,1				86.338	172.676	1.945	2.263	0.973	1.892			170.325	340.650	1.940	2.265	0.970	1.881
	Ø,1,2				58.408	175.224	2.875	3.345	0.958	2.756			114.903	344.709	2.875	3.357	0.958	2.756
	Ø,1,2,3				44.210	176.840	3.799	4.419	0.950	3.608			86.828	347.312	3.805	4.443	0.951	3.619
480	Ø	1.0	.621282682E-02	.621080399E-04	332.520	332.520	1	1.152	1	1	.121580176E-01	.121533871E-03	654.200	654.200	1	1.155	1	1
	Ø,1				170.910	341.820	1.946	2.240	0.973	1.893			336.420	672.840	1.945	2.247	0.972	1.891
	Ø,1,2				114.670	344.010	2.900	3.339	0.967	2.803			225.660	676.980	2.899	3.349	0.966	2.802
	Ø,1,2,3				86.480	345.920	3.845	4.428	0.961	3.696			170.390	681.560	3.839	4.436	0.960	3.685
960	Ø	1.0	.658917427E-02	.657439232E-04	662.550	662.550	1	1.147	1	1	.131125152E-01	.130832195E-03	1304.110	1304.110	1	1.149	1	1
	Ø,1				339.740	679.480	1.950	2.236	0.975	1.902			668.390	1336.780	1.951	2.241	0.976	1.903
	Ø,1,2				227.960	683.880	2.906	3.333	0.969	2.816			449.080	1347.240	2.904	3.336	0.968	2.811
	Ø,1,2,3				171.820	687.280	3.856	4.422	0.964	3.717			338.630	1354.520	3.851	4.424	0.963	3.708
1920	Ø	1.0	.593522564E-02	.593066216E-04	1330.143	1330.143	1	1.138	1	1	.118404701E-01	.118136406E-03	2616.143	2616.143	1	1.142	1	1
	Ø,1				677.565	1355.130	1.963	2.234	0.982	1.927			1333.985	2667.970	1.961	2.240	0.981	1.923
	Ø,1,2				454.303	1362.909	2.928	3.332	0.976	2.857			893.750	2681.250	2.927	3.343	0.976	2.856
	Ø,1,2,3				341.755	1367.020	3.892	4.430	0.973	3.787			672.883	2691.532	3.888	4.440	0.972	3.779
TIME-STEPS:			60								120							

Table IV.B.3.3-t1: Experimental Results and Performance Measurements of the Parallel Algorithm for the *G.E.C.* Method on the *'NEPTUNE'* Prototype System.

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	E_p	$F_p.T_s^{(e)}$
240	Ø	1.0	.341715552E-01	.256240368E-03	655.305	655.305	1	1.169	1	1
	Ø,1				337.360	674.720	1.942	2.271	0.971	1.887
	Ø,1,2				228.065	684.195	2.873	3.360	0.958	2.752
	Ø,1,2,3				172.603	690.412	3.797	4.439	0.949	3.604
480	Ø	1.0	.156205185E-01	.155746937E-03	1298.380	1298.380	1	1.155	1	1
	Ø,1				665.700	1331.400	1.950	2.253	0.975	1.902
	Ø,1,2				449.620	1348.860	2.888	3.336	0.963	2.780
	Ø,1,2,3				338.740	1354.960	3.833	4.429	0.958	3.673
960	Ø	1.0	.232403427E-01	.232338905E-03	2590.950	2590.950	1	1.148	1	1
	Ø,1				1327.720	2655.440	1.951	2.240	0.976	1.904
	Ø,1,2				889.940	2669.820	2.911	3.342	0.970	2.825
	Ø,1,2,3				670.560	2682.240	3.864	4.436	0.966	3.732
1920	Ø	1.0	.232702345E-01	.232219696E-03	5193.213	5193.213	1	1.143	1	1
	Ø,1				2644.630	5289.260	1.964	2.244	0.982	1.928
	Ø,1,2				1772.875	5318.625	2.929	3.347	0.976	2.860
	Ø,1,2,3				1335.070	5340.280	3.890	4.445	0.972	3.783
TIME-STEPS:			240							

Table IV.B.3.3-t1(cont.d.): Experimental Results and Performance Measurements of the Parallel Algorithm for the *G.E.C.* Method on the *'NEPTUNE'* Prototype System.

The greater the number of cooperating processors, the higher the real Cost (C_p) of the algorithm, along with the *linear* to them Speed-up values obtained, which produce slightly better peaks when the grid size is *(1920×240)* for the specified combination of 2 and 3 processors; on the other hand, when all 4 processors are cooperating the best peak is achieved for the grid of size *(1920×60)*. Again, we must note the fact that the Speed-ups achieved, and consequently the values for the Efficiency and $F_p.T_s^{(e)}$ parameters, when increasing the number of internal grid points, show a very smooth analogous increase which has not been observed with any of the previous parallel algorithms. This last observation establishes the present algorithm as the most efficient so far, but still this method, together with the *GEU* method, does suffer from the restrictive stability condition $r \leq 1$, which prevents the revelation of the potential hidden in this sort of *GE* schemes; and this potential, as it will be subsequently exemplified, having to compromise with a standard given *MIMD* 'testbed' of processors is quite unbelievable.

Finally, and in terms of the *convergence* of the solution achieved through the *GEC* method, the best accuracy obtained and subsequently the optimum grid size was for *(1920×60)*.

- Performance Analysis

In accordance with the list of parameters for the performance model and measuring formulae introduced in *(par.-IV.B.3.1)*, the program and system dependent performance analyses figures are given in *Tables (IV.B.3.3-t2,t3)*, respectively.

Again, and for all the remaining *GE* parallel algorithms, note that the parameters $Ac_{(i,j)}$ and $Tc_{(i,j)}^{(t)}$, for the *fifth* phase of the algorithms,

G_S	P_H	PROCESSORS (p)		$Ac_{(i,j)}$	$T_c^{(t)}{}_{(i,j)}$ (secs)	I_{cl}	Ac_p	$T_c^{(t)}{}_p$ (secs)	$N_{p(p)}$	$L_{O(//)}$	SHARED DATA		PARALLEL PATH		
		N_t	S_p								$R_{a(s)}$	$O^{(t)}_{st(s)}$	$R_{a(//)}$	$O^{(t)}_{st(//)}$	
240×240	3	$p \leq 2N_{B.P.}$ $[p \mid N_{B.P.}]$	$\mathcal{O}(p)$	60 flops	0.023	$\frac{480}{p}$	$\frac{28.8\times10^3}{p}$ flops	$\frac{11.088}{p}$	1	1	1:60 flops	0.003%	1:$\frac{28.8\times10^3}{p}$ flops	0.011p%	•••
	4	$p \leq N_{I.P.}$ $[p \mid N_{I.P.}]$	$\mathcal{O}(p)$	56 flops	0.022	$\frac{240}{p}$	$\frac{13.44\times10^3}{p}$ flops	$\frac{5.174}{p}$	1	1	1:56 flops	0.004%	1:$\frac{13.44\times10^3}{p}$ flops	0.023p%	
	5	$p \leq N_{I.P.}$ $[p \mid N_{I.P.}]$	$\mathcal{O}(p)$	49 flops	0.019	$\frac{120}{p}$	$\frac{5.88\times10^3}{p}$ flops	$\frac{2.264}{p}$	1	240	1:4 flops	0.049%	1:$\frac{5.88\times10^3}{p}$ flops	0.053p%	

Table IV.B.3.3-t2: A Program Dependent Performance Analysis of the Parallel Algorithm for the *G.E.C.* Method.

LIMITS TO PERFORMANCE			$Id_t^{(t)}$	W_{st} (secs)
$S_{d(r)}$	$S'_{d(r)}$	$S_{h(r)}$		
m_p=30,596	m_p=24,509	m_p=$\frac{9,240}{p}$	~1.5%	0.005
m_p=28,556	m_p=22,875	m_p=$\frac{4,312}{p}$		0.005
m_p=2,039	m_p=1,633	m_p=$\frac{1,886}{p}$		1.413

G_S	$T_s^{(e)}$ (secs)	S_p			$\sum_{i=1}^{p} c_{y_i}$	t_{cy} (μsecs)	$Id_t^{(e)}$	t_b (μsecs)	PARALLEL PATH		$T_s^{(e)}$ (secs)	PARALLEL CONTROL	$T_s^{(e)}$ (secs)	SHARED DATA	W_{dc} (secs)
	[XPFCL]	∅,1	∅,1,2	∅,1,2,3					$O^{(e)}_{st(//)}$	$O^{(e)}_{cn(//)}$	[XPFCLN]	$O^{(e)}_{tl(//)}$	[XPFCLS]	$O^{(e)}_{tl(s)}$	
240×240	655.305	1.942	2.873	3.797	2723	~10,800	~4.3%	~686	0.17%	0.27%	654.690	0.09%	653.800	0.14%	29.408

Table IV.B.3.3-t3: A System Dependent Performance Analysis of the Parallel Algorithm for the *G.E.C.* Method.

have been estimated considering the corresponding complexities per group of two internal grid points, at each time-level, the rest of the parameters computed accordingly.

The same generalization applies for the complementary information, to that given in *(Appendix C-II/par.-II.B.3.1)*, presented in *(par.-IV.B.3.1)*. In particular for the information obtained from the shared array *ITIME*, when testing the remaining *GE* parallel algorithms presented in this *Chapter*, a plethora of runs was carried out as before, always considering the average figures for every information taken from this array. This essential information for the estimation of the experiment dependent performance analysis parameters of the present algorithm and for the case that all *four* processors of the *NEPTUNE* system were cooperating was as follows:

i) The smallest run-time $T_p^{(e)}$ to be utilized in formula *(IV.B.3.1:25)* was 172.580 secs;

ii) the total number of wait cycles to be utilized in formula *(IV.B.3.1:27)* was ~2723, which implied an average number of wait cycles per processor of ~681;

iii) the average experimental timing of all cooperating processors was ~172.591 secs; and,

iv) the number of parallel paths run by each processor, considering the average of all cooperating processors but P_0, was 243.

With respect to the time the system was not used productively (W), being estimated through the formula *(IV.B.3.1:22)* by using the average experimental timing in *iii)*, it was ~35.059 secs; again a good approximation to this total wasted time can be obtained from the *sum* of the wasted times statically and dynamically given in the performance analysis *Tables*.

Finally, the same note must be made, as for the parallel algorithm for the *GEU* method, about the normalization and integer rounding of the number of accesses to shared memory, in the *fifth phase* of the algorithm, a fact which is bound to introduce slight discrepancies between the results found through the alternate cross-verifying formulae *(IV.B.3.1:23,24)* for the wasted time statically.

The seeming identification of *Table (IV.B.3.3-t2)* to *Table (IV.B.3.2-t2)* is due to the fact that the program dependent performance analysis is only concerned with the *flops* of the algorithms, ignoring all integer operations, shared and local transfers, etc. The differences between the algorithms become apparent examining all the experimentally estimated parameters.

To conclude, the real processing-to-access ratio, in the *fifth phase* of this algorithm, was again, as for the previous algorithm, 49 *flops* over 12 accesses to the shared data resource, for each implementation cycle.

IV.B.3.4: The '(Single) Alternating Group Explicit' - *(S)AGE* Method: Experimental Results And Performance Analysis On The '*NEPTUNE*' Prototype System

This scheme results from the coupled use of the *GEU* and *GEC* schemes at each alternate time-level. In correspondence with the formulae in *(IV.B.3:30)* it is given by

$$\left.\begin{aligned}(I+r\hat{G}_{1,j})\underline{U}_{j+1} &= (I-r\hat{G}_{2,j})\underline{U}_{j} + \underline{b}_{3}\\ (I+r\hat{G}_{2,j})\underline{U}_{j+2} &= (I-r\hat{G}_{1,j})\underline{U}_{j+1} + \underline{b}_{4}\end{aligned}\right\};\ \textit{(IV.B.3.4:1)}$$

the *brick* diagram for this scheme is as in *Figure (IV.B.3.4-f1)*.

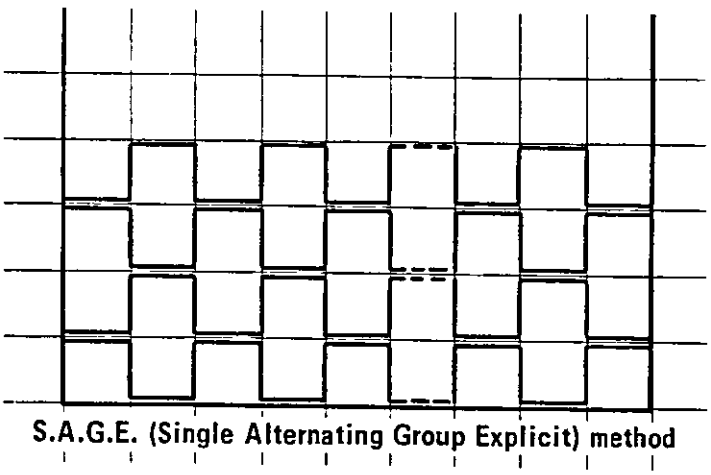


Figure IV.B.3.4-f1: The Representative Diagram of this Scheme.

The implemented program of this method, with a similar as before six-phase formation, is included in *Appendix C-IV* under the name *MB$5.SAGEONI*†. Again, in a comparative manner and in terms of the differences occurring from the previous algorithms, in the *second phase*, the exact theoretical values are computed at the internal points only, at the maximum time-level, using the exact solution formula *(IV.B.3.1:2)*. In the *third phase*, the computation of the exact values, according to this formula, at all the points on both boundaries, for all time-levels, starts from time-level *one* up to the maximum time-level. In particular for the complicated programming *fifth phase*, the approximate values at the internal points, at every time-level and for all time-steps, are computed using the Group Explicit finite-difference and Saul'yev's asymmetric formulae in accordance with the alternate formation of this scheme.

†*It stands for Single Alternating Group Explicit method for Odd Number of Intervals.*

The implementation's *granularity factor*, the parallel constructs and shared/non-shared arrays utilized in the program have remained the same as for the previous parallel algorithms, while, to be specific, the arrays have been declared for similar purposes as for the *MB$5.GEUONI* program.

- Experimental Results

Since this scheme, on the same region of the *open rectangle* $[0,1]\times[0,+\infty)$, has been proved to be unconditionally stable for all $r>0$ (see Abdullah [*ABDU83*]) enabled us to use greater values for the grid ratio, retaining always the grid size correspondence with all the previous parallel algorithms. In fact, according to the maximum grid size allowed by the *NEPTUNE* system for the deterministic Standard Explicit method, we have experimented with grid ratio values of 1,2 and 4, respectively; the grid sizes[†] experimented with, along with the results obtained, are being presented in *Tables (IV.B.3.4-t1,t4,t5)*, accordingly.

The concepts of the parameters appearing in these *Tables* are as they were presented in the previous *GE* parallel algorithms, with the exception of a new introduced parameter $\bar{R}_{S_p}$ in *Tables (IV.B.3.4-t4,t5)*, which mirrors the *Reference internal Speed-up* of the algorithm depending on the values for r; in other words, it provides the ratio between the experimental time-complexity of the uniprocessor *basis* solution and the experimental time-complexities achieved, in terms of r, in a uniprocessor and parallel implementation of the same algorithm.

To draw some conclusions by comparing the results of *Table (IV.B.3.4-t1)* with the corresponding ones in the *Tables* of the previous

[†] *Seeking always a 'balanced' implementation in respect of the utilized processors each time.*

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	E_p	$F_p.T_s^{(e)}$	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	E_p	$F_p.T_s^{(e)}$
240	Ø	1.0	.119335875E-01	.866055489E-04	175.248	175.248	1	1.115	1	1	.239556655E-01	.175774097E-03	345.300	345.300	1	1.117	1	1
	Ø,1				89.860	179.720	1.950	2.174	0.975	1.902			177.320	354.640	1.947	2.175	0.974	1.896
	Ø,1,2				60.778	182.334	2.883	3.215	0.961	2.771			119.660	358.980	2.886	3.224	0.962	2.776
	Ø,1,2,3				45.948	183.792	3.814	4.252	0.954	3.637			90.420	361.680	3.819	4.266	0.955	3.646
480	Ø	1.0	.599190593E-02	.599026680E-04	348.000	348.000	1	1.100	1	1	.111145377E-01	.111103058E-03	684.570	684.570	1	1.104	1	1
	Ø,1				178.350	356.700	1.951	2.147	0.976	1.904			351.420	702.840	1.948	2.151	0.974	1.897
	Ø,1,2				119.700	359.100	2.907	3.199	0.969	2.817			235.800	707.400	2.903	3.205	0.968	2.809
	Ø,1,2,3				90.280	361.120	3.855	4.241	0.964	3.715			178.420	713.680	3.837	4.236	0.959	3.680
960	Ø	1.0	.643055141E-02	.641942024E-04	694.620	694.620	1	1.094	1	1	.126221851E-01	.126004219E-03	1366.180	1366.180	1	1.097	1	1
	Ø,1				355.440	710.880	1.954	2.137	0.977	1.910			695.190	1390.380	1.965	2.155	0.983	1.931
	Ø,1,2				238.020	714.060	2.918	3.192	0.973	2.839			467.260	1401.780	2.924	3.206	0.975	2.850
	Ø,1,2,3				179.180	716.720	3.877	4.240	0.969	3.757			352.600	1410.400	3.875	4.249	0.969	3.753
1920	Ø	1.0	.633467734E-02	.633001328E-04	1391.230	1391.230	1	1.088	1	1	.121325105E-01	.121235847E-03	2738.860	2738.860	1	1.091	1	1
	Ø,1				705.080	1410.160	1.973	2.147	0.987	1.947			1385.270	2770.540	1.977	2.157	0.989	1.955
	Ø,1,2				473.460	1420.380	2.938	3.197	0.979	2.878			930.190	2790.570	2.944	3.212	0.981	2.890
	Ø,1,2,3				356.090	1424.360	3.907	4.251	0.977	3.816			701.420	2805.680	3.905	4.260	0.976	3.812
TIME-STEPS:			60								120							

Table IV.B.3.4-t1: Experimental Results and Performance Measurements of the Parallel Algorithm for the *(S).A.G.E.* Method (for r=1) on the *'NEPTUNE'* Prototype System.

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	RS_p	E_p	$F_p.T_s^{(e)}$
240	Ø	1.0	.452918150E-01	.339627266E-03	685.175	685.175	1	1.118	1	1
	Ø,1				351.200	702.400	1.951	2.182	0.975	1.903
	Ø,1,2				237.675	713.025	2.883	3.224	0.961	2.770
	Ø,1,2,3				180.140	720.560	3.804	4.254	0.951	3.617
480	Ø	1.0	.219819658E-01	.219762325E-03	1360.020	1360.020	1	1.103	1	1
	Ø,1				698.020	1396.040	1.948	2.149	0.974	1.898
	Ø,1,2				469.340	1408.020	2.898	3.196	0.966	2.799
	Ø,1,2,3				354.580	1418.320	3.836	4.231	0.959	3.678
960	Ø	1.0	.193599276E-01	.193238258E-03	2712.530	2712.530	1	1.097	1	1
	Ø,1				1380.620	2761.240	1.965	2.155	0.982	1.930
	Ø,1,2				928.930	2786.790	2.920	3.202	0.973	2.842
	Ø,1,2,3				699.610	2798.440	3.877	4.252	0.969	3.758
1920	Ø	1.0	.232282057E-01	.232040882E-03	5435.620	5435.620	1	1.092	1	1
	Ø,1				2765.210	5530.420	1.966	2.146	0.983	1.932
	Ø,1,2				1853.370	5560.110	2.933	3.202	0.978	2.867
	Ø,1,2,3				1393.490	5573.960	3.901	4.258	0.975	3.804
TIME-STEPS:			240							

Table IV.B.3.4-t1(cont.d.): Experimental Results and Performance Measurements of the Parallel Algorithm for the *(S).A.G.E.* Method (for r=1) on the *'NEPTUNE'* Prototype System.

methods, the impression given is that the present algorithm, in terms of the time-complexities and *Relative Speed-ups* (R_{S_p}) achieved running it, behaves worse than the *GEC* method and better than the *GEU* one; certainly, there is not any comparison with the *standard* algorithm's time-complexities.

The values for the S_p and E_p parameters in this *Table* prove in general to be better than the corresponding values for the *GEC* method and are more or less equivalent, or even better (especially as the size of the grid increases) than those for the *GEU* method. In fact, the Speed-up and Efficiency values show almost the smooth and analogous to the number of internal grid points increase observed in the algorithm for the *GEC* method, while the peaks observed (i.e. for 2 and 3 cooperating processors for the grid of size (*1920×120*) and when all 4 processors are utilized for the grid of size (*1920×60*)) are much higher than the highest peaks of the *GEU* method. Apparently this is due to the excessive code involved as was for the latter method and these results should not be considered seriously by the implementor.

The tremendous potential of this method unfolds when passing the restrictive barrier of *1* for the grid ratio and experiment with greater values. In general, from every aspect, either numerical or experimental, this method performs unbelievably. Although it is a matter of less interest to discuss the values of S_p and E_p parameters, however, a very close fluctuation in their values has been observed reaching instances where they prove again to be even better than those of the *GEU* method. The revealing factors of the tremendous potential of the method are the R_{S_p} and $\bar{R}_{S_p}$ parameters. More analytically, by examining *Tables (IV.B.3.4-t4,t5)* we note that the *Relative Speed-ups* are amazingly

keeping-up an analogous increase in values according to the increase of the grid ratio r, always being $>O(r.p)$. In actual fact, for r=4, we reached a maximum 16.766 times acceleration comparing to the *standard* algorithm by only using 4 processors; it is really a very interesting matter what the result would be if we could experiment with this method for greater grid sizes (and consequently $r's$), unfortunately not feasible on the *NEPTUNE* prototype system. In other words, we have proved that this method can really accelerate in a manner independent of the system's restrictive, in terms of the number of available processors, configuration.

The explanation for this amazing result lies on the $\bar{R}_{S_p}$ parameter which exhibits a *Reference internal acceleration* of $O(r.p)$, reaching a maximum value of 15.357, again, for r=4 and in a maximum utilization of the system, due to the smaller time-complexities (and consequently real Costs) caused by the proportional reduction in the total number of time-steps.

Finally, from the pure numerical point of view, any possible doubt about the accuracy of the solution obtained through this method is proved unecessary, since, by examining *Tables (IV.B.3.4-t1,t4,t5)* correspondingly, we note that the accuracy of the method generally increases along with the grid ratio r, to reach a value for the maximum absolute error (when r=4)† which is the smallest achieved from the parallel algorithms experimented with to-date.

- Performance Analysis

The program and system dependent performance analyses concerning

†*For the maximum grid size indirectly allowed by the system.*

the basic and maximal experimental cases allowed by the system, in terms of the grid ratio (i.e. when r=1 and r=4, respectively) and in accordance with the parameters of the performance model and measuring formulae presented in (*par.-IV.B.3.1*), are given in *Tables (IV.B.3.4-t2,t3,t6,t7)*, respectively.

In addition to the general notes made in the performance analysis part of the parallel algorithm for the *GEC* method, applicable to all *GE* algorithms presented in this *Chapter*, the particularly essential information for the estimation of the experiment dependent performance analysis parameters of the present algorithm, obtained from the shared array *ITIME* in the same manner as for all previous algorithms and in a maximum system utilization, was as follows:

i) The smallest run-time $T_p^{(e)}$ to be utilized in formula *(IV.B.3.1:25)*, when r=1, was 180.125 secs., while when r=4, was 45.718 secs;

ii) the total number of wait cycles to be utilized in formula *(IV.B.3.1:27)*, when r=1, was ~2692, while when r=4, was ~716, which implied average numbers of wait cycles per processor of ~673 and ~179, respectively;

iii) the average experimental timings of all cooperating processors for these cases were ~180.132 secs and ~45.722 secs, accordingly; and,

iv) the number of parallel paths run by each processor, considering the average of all cooperating processors but P_0, in correspondence with the values of the grid ratio were 243 and 63, respectively.

G_S	P_H	PROCESSORS (p)		$Ac_{(i,j)}$	$Tc^{(t)}_{(i,j)}$ (secs)	I_{cl}	Ac_p	$Tc^{(t)}_p$ (secs)	$N_{p(p)}$	$L_{O(///)}$	SHARED DATA		PARALLEL PATH	
		N_t	S_p								$R_{a(s)}$	$O^{(t)}_{st(s)}$	$R_{a(///)}$	$O^{(t)}_{st(///)}$
240×240	3	$p \leq 2N_{B.P.}$ $[p \mid N_{B.P.}]$	$O(p)$	60 flops	0.023	$\frac{480}{p}$	$\frac{28.8\times10^3}{p}$ flops	$\frac{11.088}{p}$	1	1	1:60 flops	0.003%	1:$\frac{28.8\times10^3}{p}$ flops	0.011p%
	4	$p \leq N_{I.P.}$ $[p \mid N_{I.P.}]$	$O(p)$	56 flops	0.022	$\frac{240}{p}$	$\frac{13.44\times10^3}{p}$ flops	$\frac{5.174}{p}$	1	1	1:56 flops	0.004%	1:$\frac{13.44\times10^3}{p}$ flops	0.023p%
	5	$p \leq N_{I.P.}$ $[p \mid N_{I.P.}]$	$O(p)$	49 flops	0.019	$\frac{120}{p}$	$\frac{5.88\times10^3}{p}$ flops	$\frac{2.264}{p}$	2	120	1:4 flops	0.049%	1:$\frac{5.88\times10^3}{p}$ flops	0.053p%

Table IV.B.3.4-t2: A Program Dependent Performance Analysis of the Parallel Algorithm for the *(S).A.G.E.* Method (for r=1).

LIMITS TO PERFORMANCE			$Id^{(t)}_t$	W_{st} (secs)
$S_{d(r)}$	$S'_{d(r)}$	$S_{h(r)}$		
m_p=30,596	m_p=24,509	$m_p=\frac{9,240}{p}$	~1.5%	0.005
m_p=28,556	m_p=22,875	$m_p=\frac{4,312}{p}$		0.005
m_p=2,039	m_p=1,633	$m_p=\frac{1,886}{p}$		1.413

G_S	$T^{(e)}_s$ (secs)	S_p			$\sum_{i=1}^{p} c_{y_i}$	t_{cy} (μsecs)	$Id^{(e)}_t$	t_b (μsecs)	PARALLEL PATH		$T^{(e)}_s$ (secs)	PARALLEL CONTROL	$T^{(e)}_s$ (secs)	SHARED DATA	W_{dc} (secs)
	[XPFCL]	Ø,1	Ø,1,2	Ø,1,2,3					$O^{(e)}_{st(///)}$	$O^{(e)}_{cn(///)}$	[XPFCLN]	$O^{(e)}_{tl(///)}$	[XPFCLS]	$O^{(e)}_{tl(s)}$	
240×240	685.175	1.951	2.883	3.804	2692	~10,800	~4%	~686	0.16%	0.26%	684.450	0.11%	683.967	0.07%	29.074

Table IV.B.3.4-t3: A System Dependent Performance Analysis of the Parallel Algorithm for the *(S).A.G.E.* Method (for r=1).

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	$\bar{R}_{S_p}$	E_p	$F_p.T_s^{(e)}$
240	Ø	2.0	.229158811E-01	.168144703E-03	173.960	173.960	1	2.217	1.985	1	1
	Ø,1				89.040	178.080	1.954	4.332	3.878	0.977	1.909
	Ø,1,2				60.200	180.600	2.890	6.408	5.736	0.963	2.783
	Ø,1,2,3				45.570	182.280	3.817	8.465	7.577	0.954	3.643
480	Ø	2.0	.602386892E-02	.600218773E-04	346.020	346.020	1	2.184	1.978	1	1
	Ø,1				177.500	355.000	1.949	4.258	3.857	0.975	1.900
	Ø,1,2				119.170	357.510	2.904	6.342	5.744	0.968	2.810
	Ø,1,2,3				89.510	358.040	3.866	8.444	7.648	0.966	3.736
960	Ø	2.0	.573427230E-02	.573396683E-04	687.910	687.910	1	2.178	1.986	1	1
	Ø,1				350.900	701.800	1.960	4.269	3.893	0.980	1.922
	Ø,1,2				235.550	706.650	2.920	6.360	5.800	0.973	2.843
	Ø,1,2,3				177.970	711.880	3.865	8.418	7.676	0.966	3.735
1920	Ø	2.0	.531333312E-02	.530481339E-04	1375.810	1375.810	1	2.172	1.991	1	1
	Ø,1				698.990	1397.980	1.968	4.275	3.918	0.984	1.937
	Ø,1,2				471.620	1414.860	2.917	6.335	5.807	0.972	2.837
	Ø,1,2,3				354.920	1419.680	3.876	8.418	7.717	0.969	3.757
TIME-STEPS:			60								

Table IV.B.3.4-t4: Experimental Results and Performance Measurements of the Parallel Algorithm for the *(S).A.G.E.* Method (for r=2) on the *'NEPTUNE'* Prototype System.

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	$\bar{R}_{S_p}$	E_p	$F_p.T_s^{(e)}$
240	Ø	2.0	.438451469E-01	.328779221E-03	343.953	343.953	1	2.228	1.992	1	1
	Ø,1				176.318	352.636	1.951	4.346	3.886	0.975	1.903
	Ø,1,2				118.973	356.919	2.891	6.441	5.759	0.964	2.786
	Ø,1,2,3				90.068	360.272	3.819	8.508	7.607	0.955	3.646
480	Ø	2.0	.117268600E-01	.117063522E-03	679.180	679.180	1	2.209	2.002	1	1
	Ø,1				347.780	695.560	1.953	4.313	3.911	0.976	1.907
	Ø,1,2				233.540	700.620	2.908	6.423	5.823	0.969	2.819
	Ø,1,2,3				176.780	707.120	3.842	8.486	7.693	0.960	3.690
960	Ø	2.0	.961446390E-02	.961422920E-04	1353.130	1353.130	1	2.198	2.005	1	1
	Ø,1				689.260	1378.520	1.963	4.316	3.935	0.982	1.927
	Ø,1,2				464.440	1393.320	2.913	6.405	5.840	0.971	2.829
	Ø,1,2,3				351.010	1404.040	3.855	8.474	7.728	0.964	3.715
1920	Ø	2.0	.899305195E-02	.898241997E-04	2710.050	2710.050	1	2.190	2.006	1	1
	Ø,1				1375.670	2751.340	1.970	4.314	3.951	0.985	1.940
	Ø,1,2				927.640	2782.920	2.921	6.397	5.860	0.974	2.845
	Ø,1,2,3				698.720	2794.880	3.879	8.493	7.779	0.970	3.761
TIME-STEPS:			120								

Table IV.B.3.4-t4(cont.d.): Experimental Results and Performance Measurements of the Parallel Algorithm for the *(S).A.G.E.* Method (for r=2) on the *'NEPTUNE'* Prototype System.

$N_{I.P}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	$\bar{R}_{S_p}$	E_p	$F_p.T_s^{(e)}$
240	Ø	4.0	.447672009E-01	.335693359E-03	174.062	174.062	1	4.402	3.936	1	1
	Ø,1				89.356	178.712	1.948	8.575	7.668	0.974	1.897
	Ø,1,2				60.420	181.260	2.881	12.682	11.340	0.960	2.766
	Ø,1,2,3				45.726	182.904	3.807	16.758	14.984	0.952	3.623
480	Ø	4.0	.631995499E-02	.631213188E-04	344.070	344.070	1	4.360	3.953	1	1
	Ø,1				177.190	354.380	1.942	8.466	7.675	0.971	1.885
	Ø,1,2				119.370	358.110	2.882	12.567	11.393	0.961	2.769
	Ø,1,2,3				90.240	360.960	3.813	16.624	15.071	0.953	3.634
960	Ø	4.0	.657623261E-02	.654459000E-04	686.320	686.320	1	4.334	3.952	1	1
	Ø,1				350.720	701.440	1.957	8.481	7.734	0.978	1.915
	Ø,1,2				235.620	706.860	2.913	12.625	11.512	0.971	2.828
	Ø,1,2,3				177.770	711.080	3.861	16.733	15.259	0.965	3.726
1920	Ø	4.0	.512412935E-02	.507831573E-04	1374.880	1374.880	1	4.316	3.954	1	1
	Ø,1				699.810	1399.620	1.965	8.480	7.767	0.982	1.930
	Ø,1,2				469.720	1409.160	2.927	12.633	11.572	0.976	2.856
	Ø,1,2,3				353.940	1415.760	3.885	16.766	15.357	0.971	3.772
TIME-STEPS:			60								

Table IV.B.3.4-t5: Experimental Results and Performance Measurements of the Parallel Algorithm for the *(S).A.G.E.* Method (for r=4) on the *'NEPTUNE'* Prototype System.

G_S	P_H	PROCESSORS (p)		$Ac_{(i,j)}$	$T_{c(i,j)}^{(t)}$ (secs)	I_{cl}	Ac_p	$T_{c_p}^{(t)}$ (secs)	$N_{p(p)}$	$L_{O(//)}$	SHARED DATA		PARALLEL PATH	
		N_t	S_p								$R_{a(s)}$	$O_{st(s)}^{(t)}$	$R_{a(//)}$	$O_{st(//)}^{(t)}$
240×60	3	$p \lesssim 2N_{B.P.}$ $[p \mid N_{B.P.}]$	$O(p)$	60 flops	0.023	$\frac{120}{p}$	$\frac{7.2\times10^3}{p}$ flops	$\frac{2.772}{p}$	1	1	1:60 flops	0.003%	$1:\frac{7.2\times10^3}{p}$ flops	0.043p%
	4	$p \lesssim N_{I.P.}$ $[p \mid N_{I.P.}]$	$O(p)$	56 flops	0.022	$\frac{240}{p}$	$\frac{13.44\times10^3}{p}$ flops	$\frac{5.174}{p}$	1	1	1:56 flops	0.004%	$1:\frac{13.44\times10^3}{p}$ flops	0.023p%
	5	$p \lesssim N_{I.P.}$ $[p \mid N_{I.P.}]$	$O(p)$	49 flops	0.019	$\frac{120}{p}$	$\frac{5.88\times10^3}{p}$ flops	$\frac{2.264}{p}$	2	30	1:4 flops	0.049%	$1:\frac{5.88\times10^3}{p}$ flops	0.053p%

Table IV.B.3.4-t6: A Program Dependent Performance Analysis of the Parallel Algorithm for the *(S).A.G.E.* Method (for r=4).

LIMITS TO PERFORMANCE			$Id_t^{(t)}$	W_{st} (secs)
$S_{d(r)}$	$S'_{d(r)}$	$S_{h(r)}$		
m_p=30,596	m_p=24,509	$m_p=\frac{2,310}{p}$	~1.5%	0.005
m_p=28,556	m_p=22,875	$m_p=\frac{4,312}{p}$		0.005
m_p=2,039	m_p=1,633	$m_p=\frac{1,886}{p}$		0.353

G_S	$T_s^{(e)}$ (secs)	S_p			$\sum_{i=1}^{p} c_{y_i}$	t_{cy} (μsecs)	$Id_t^{(e)}$	t_b (μsecs)	PARALLEL PATH		$T_s^{(e)}$ (secs)	PARALLEL CONTROL	$T_s^{(e)}$ (secs)	SHARED DATA	W_{dc} (secs)
	[XPFCL]	∅,1	∅,1,2	∅,1,2,3					$O_{st(//)}^{(e)}$	$O_{cn(//)}^{(e)}$	[XPFCLN]	$O_{tl(//)}^{(e)}$	[XPFCLS]	$O_{tl(s)}^{(e)}$	
240×60	174.062	1.948	2.881	3.807	716	~10,800	~4.2%	~686	0.17%	0.27%	173.880	0.1%	173.690	0.11%	7.733

Table IV.B.3.4-t7: A System Dependent Performance Analysis of the Parallel Algorithm for the *(S).A.G.E.* Method (for r=4).

With reference to the times the system was not used productively (W), being again estimated through the formula *(IV.B.3.1:22)* by using the average experimental timings in *iii)*, accordingly, they were ~35.353 secs and ~8.826 secs; once more, good approximations to these total wasted times can be obtained from the *sum* of the respective wasted times statically and dynamically, presented in the performance analysis *Tables*.

To conclude, the reader should bear in mind the notices given for the previous *GE* algorithms about the ratio of accesses to the shared data structure, to avoid any illusive discrepancies when seeking the cross-verification of the results obtained through the alternate formulae *(IV.B.3.1:23,24)* for the wasted time statically, in the *fifth phase* of the algorithm. Again, the real processing-to-access ratio, for this phase, was 49 *flops* over 12 accesses to the shared data resource, for each implementation cycle.

IV.B.3.5: The '(Double) Alternating Group Explicit' - *(D)AGE* Method: Experimental Results And Performance Analysis On The *'NEPTUNE'* Prototype System

This scheme, as we have discussed in (*par.-IV.B.3*), is a periodic rotation of the *(S)AGE* two time-level scheme resulting in a *four* time-level step process with the second half cycle in reverse order. In correspondence with the formulae in *(IV.B.3:31)* it is given by

$$\left.\begin{aligned}
(I+r\hat{G}_{1,j})\underline{U}_{j+1} &= (I-r\hat{G}_{2,j})\underline{U}_{j} + \underline{b}_3 \\
(I+r\hat{G}_{2,j})\underline{U}_{j+2} &= (I-r\hat{G}_{1,j})\underline{U}_{j+1} + \underline{b}_4 \\
(I+r\hat{G}_{2,j})\underline{U}_{j+3} &= (I-r\hat{G}_{1,j})\underline{U}_{j+2} + \underline{b}_4 \\
(I+r\hat{G}_{1,j})\underline{U}_{j+4} &= (I-r\hat{G}_{2,j})\underline{U}_{j+3} + \underline{b}_3
\end{aligned}\right\} \quad ; \qquad (IV.B.3.5:1)$$

the *brick* diagram for this scheme is as in *Figure (IV.B.3.5-f1)*.

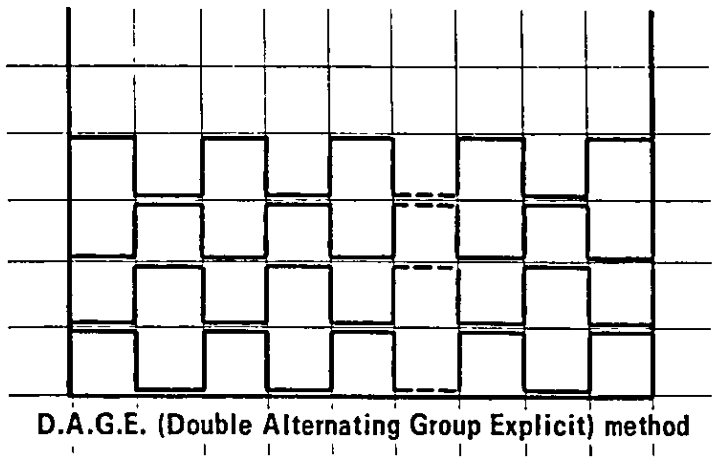


Figure IV.B.3.5-f1: The Representative Diagram of this Scheme.

Again, this method has been implemented in a six-phase formatted program which is included in *Appendix C-IV* under the name *MB$5.DAGEONI*[†]. More specifically, there is a close correspondence between this implementation and the implementation of the *(S)AGE* method for all phases of the algorithm, but the *fifth*. In there, due to the periodic use of the *GEU* and *GEC* basic schemes synthesizing this scheme, a *flag* is introduced to control their alternating sequence and form this *four* time-level step process. Any other program structuring specifications made for the parallel algorithm for the *(S)AGE* method are well applied for the present algorithm.

- Experimental Results

In a similar manner, as for the previous *GE* scheme, it has been proved that the present scheme is also unconditionally stable for all $r>0$ (see Abdullah [*ABDU83*]), on the same region of the *open rectangle* $[0,1]\times[0,+\infty)$.

[†]*It stands for Double Alternating Group Explicit method for Odd Number of Intervals.*

Again for direct comparison reasons, as well as due to the system's hardware limitations, we have experimented with the same values for r as for the previous parallel algorithm. The grid sizes experimented with, along with the results obtained, are being presented in *Tables (IV.B.3.5-t1,t4,t5)*, accordingly, while for the concepts of all the involved parameters in these *Tables* the reader should refer to *(par.-IV.B.3.1)*.

To draw some conclusions by comparing the corresponding results achieved through the present algorithm, to those achieved through the algorithm for the *(S)AGE* method, we note that this method behaves more or less similarly to the latter one, while in particular with the achieved time-complexities, in certain instances and despite the continuous conditional branching due to the alternating use of the basic *GEU* and *GEC* schemes, they prove to be even smaller. This slightly affects the internal acceleration and efficiency of the method, which therefore exhibits smaller performance peaks, still retaining, though, the almost smooth and analogous to the number of internal grid points increase observed in the previous algorithm. In fact the best Speed-ups and Efficiencies achieved[†] were for the grid of size *(1920×240)* for the case of 2 and 3 cooperating processors and for the grid of size *(1920×120)* in a maximum system utilization.

Certainly, again, there is not any comparison with the *standard* algorithm from every aspect, while, in general, corresponding observations as for the case of the *(S)AGE* method, in concern with the basic schemes above, can be applied.

In a similar manner to that algorithm, the tremendous potential of

[†]*For the particular case that r=1.*

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	E_p	$F_p.T_s^{(e)}$	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	E_p	$F_p.T_s^{(e)}$
240	Ø	1.0	.119910762E-01	.870227814E-04	175.255	175.255	1	1.115	1	1	.239394195E-01	.175654888E-03	345.830	345.830	1	1.115	1	1
	Ø,1				89.893	179.786	1.950	2.174	0.975	1.900			177.080	345.160	1.953	2.178	0.976	1.907
	Ø,1,2				60.835	182.505	2.881	3.212	0.960	2.766			119.940	359.820	2.883	3.216	0.961	2.771
	Ø,1,2,3				45.963	183.852	3.813	4.251	0.953	3.635			90.800	363.200	3.809	4.248	0.352	3.627
480	Ø	1.0	.608729944E-02	.608563423E-04	346.850	346.850	1	1.104	1	1	.110959783E-01	.110924244E-03	685.660	685.660	1	1.102	1	1
	Ø,1				177.310	354.620	1.956	2.160	0.978	1.913			350.370	700.740	1.957	2.157	0.978	1.915
	Ø,1,2				119.590	358.770	2.900	3.202	0.967	2.804			235.210	705.630	2.915	3.213	0.972	2.833
	Ø,1,2,3				90.260	361.040	3.843	4.242	0.961	3.692			177.640	710.560	3.860	4.255	0.965	3.725
960	Ø	1.0	.643055141E-02	.641942024E-04	692.270	692.270	1	1.097	1	1	.126221851E-01	.126004219E-03	1364.370	1364.370	1	1.098	1	1
	Ø,1				354.770	709.540	1.951	2.141	0.976	1.904			695.700	1391.400	1.961	2.153	0.981	1.923
	Ø,1,2				237.950	713.850	2.909	3.193	0.970	2.821			466.510	1399.530	2.925	3.211	0.975	2.851
	Ø,1,2,3				178.570	714.280	3.877	4.255	0.969	3.757			351.830	1407.320	3.878	4.258	0.969	3.760
1920	Ø	1.0	.633467734E-02	.633001328E-04	1387.540	1387.540	1	1.091	1	1	.121325105E-01	.121235847E-03	2735.670	2735.670	1	1.092	1	1
	Ø,1				707.810	1415.620	1.960	2.139	0.980	1.921			1394.490	2788.980	1.962	2.143	0.981	1.924
	Ø,1,2				474.220	1422.660	2.926	3.192	0.975	2.854			934.780	2804.340	2.927	3.196	0.976	2.855
	Ø,1,2,3				356.350	1425.400	3.894	4.248	0.973	3.790			702.050	2808.200	3.897	4.256	0.974	3.796
TIME-STEPS:			60								120							

Table IV.B.3.5-t1: Experimental Results and Performance Measurements of the Parallel Algorithm for the *(D).A.G.E.* Method (for r=1) on the *'NEPTUNE'* Prototype System.

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	E_p	$F_p.T_s^{(e)}$
240	Ø	1.0	.451487377E-01	.338554382E-03	684.933	684.933	1	1.119	1	1
	Ø,1				351.053	702.106	1.951	2.183	0.976	1.903
	Ø,1,2				237.688	713.064	2.882	3.224	0.961	2.768
	Ø,1,2,3				180.180	720.720	3.801	4.253	0.950	3.613
480	Ø	1.0	.219819658E-01	.219762325E-03	1358.420	1358.420	1	1.104	1	1
	Ø,1				694.850	1389.700	1.955	2.159	0.977	1.911
	Ø,1,2				468.680	1406.040	2.898	3.201	0.966	2.800
	Ø,1,2,3				353.980	1415.920	3.838	4.238	0.959	3.682
960	Ø	1.0	.193599276E-01	.193238258E-03	2708.230	2708.230	1	1.098	1	1
	Ø,1				1381.580	2763.160	1.960	2.153	0.980	1.921
	Ø,1,2				930.690	2792.070	2.910	3.196	0.970	2.823
	Ø,1,2,3				699.290	2797.160	3.873	4.254	0.968	3.750
1920	Ø	1.0	.231327377E-01	.231087208E-03	5428.920	5428.920	1	1.093	1	1
	Ø,1				2753.400	5506.800	1.972	2.155	0.986	1.944
	Ø,1,2				1847.880	5543.640	2.938	3.211	0.979	2.877
	Ø,1,2,3				1394.440	5577.760	3.893	4.256	0.973	3.789
TIME-STEPS:			240							

Table IV.B.3.5-t1(cont.d.): Experimental Results and Performance Measurements of the Parallel Algorithm for the *(D).A.G.E.* Method (for r=1) on the *'NEPTUNE'* Prototype System.

the present method unfolds when passing the restrictive barrier of *1* for the grid ratio and experiment with greater values. From every aspect, either numerical or experimental, this method, in accordance with the previous one, performs in an unbelievable manner, which is again underlined by the R_{S_p} and $\bar{R}_{S_p}$ parameters of the performance model, exhibiting Speed-ups of $>O(r.p)$ and $O(r.p)$, respectively. To become more specific, for r=4, we achieved an even greater maximum, *Relative* to the *standard* algorithm, *acceleration* of 16.786 times, while the *Reference internal acceleration* exhibited a lower maximum, than that of the *(S)AGE* method, of 15.315 times, always in a total utilization of the system.

Finally, from the pure numerical point of view, the accuracy of this method generally increases along with the grid ratio r, reaching a maximum absolute error (when r=4)† which coincides with the smallest ever error achieved with the parallel algorithm for the *(S)AGE* method.

- Performance Analysis

The program and system dependent performance analyses, again, concerning the basic and maximal experimental cases allowed by the system, in a similar manner as for the parallel algorithm for the *(S)AGE* method, are presented in *Tables (IV.B.3.5-t2,t3,t6,t7)*, respectively.

With respect to the complementary information required for the estimation of the experiment dependent parameters appearing in these *Tables*, obtained from the shared array *ITIME* following the same procedure as for all previous *GE* parallel algorithms and in a maximum system utilization, was as follows:

†*For the maximum grid size indirectly allowed by the system.*

G_S	P_H	PROCESSORS (p)		$Ac_{(i,j)}$	$Tc^{(t)}_{(i,j)}$ (secs)	I_{cl}	Ac_p	$Tc^{(t)}_p$ (secs)	$N_{p(p)}$	$I_{d(//)}$	SHARED DATA		PARALLEL PATH	
		N_t	S_p								$R_{a(s)}$	$O^{(t)}_{st(s)}$	$R_{a(//)}$	$O^{(t)}_{st(//)}$
240×240	3	$p \leq 2N_{B.P.}$ $[p \mid N_{B.P.}]$	$O(p)$	60 flops	0.023	$\frac{480}{p}$	$\frac{28.8\times10^3}{p}$ flops	$\frac{11.088}{p}$	1	1	1:60 flops	0.003%	1:$\frac{28.8\times10^3}{p}$ flops	0.011p%
	4	$p \leq N_{I.P.}$ $[p \mid N_{I.P.}]$	$O(p)$	56 flops	0.022	$\frac{240}{p}$	$\frac{13.44\times10^3}{p}$ flops	$\frac{5.174}{p}$	1	1	1:56 flops	0.004%	1:$\frac{13.44\times10^3}{p}$ flops	0.023p%
	5	$p \leq N_{I.P.}$ $[p \mid N_{I.P.}]$	$O(p)$	49 flops	0.019	$\frac{120}{p}$	$\frac{5.88\times10^3}{p}$ flops	$\frac{2.264}{p}$	4	60	1:4 flops	0.049%	1:$\frac{5.88\times10^3}{p}$ flops	0.053p%

Table IV.B.3.5-t2: A Program Dependent Performance Analysis of the Parallel Algorithm for the *(D).A.G.E.* Method (for r=1).

LIMITS TO PERFORMANCE			$Id^{(t)}_t$	W_{st} (secs)
$S_{d(r)}$	$S'_{d(r)}$	$S_{h(r)}$		
m_p=30,596	m_p=24,509	$m_p=\frac{9.240}{p}$	~1.5%	0.005
m_p=28,556	m_p=22,875	$m_p=\frac{4,312}{p}$		0.005
m_p=2,039	m_p=1,633	$m_p=\frac{1,886}{p}$		1.413

G_S	$T^{(e)}_s$ (secs)	S_p			$\sum_{i=1}^{p} c_{y_1}$	t_{cy} (μsecs)	$Id^{(e)}_t$	t_b (μsecs)	PARALLEL PATH		$T^{(e)}_s$ (secs)	PARALLEL CONTROL	$T^{(e)}_s$ (secs)	SHARED DATA	W_{dc} (secs)
	[XPFCL]	∅,1	∅,1,2	∅,1,2,3					$O^{(e)}_{st(//)}$	$O^{(e)}_{cn(//)}$	[XPFCLN]	$O^{(e)}_{tl(//)}$	[XPFCLS]	$O^{(e)}_{tl(s)}$	
240×240	684.933	1.951	2.882	3.801	2710	~10,800	~4.1%	~ 686	0.16%	0.26%	684.330	0.09%	683.860	0.07%	29.268

Table IV.B.3.5-t3: A System Dependent Performance Analysis of the Parallel Algorithm for the *(D).A.G.E.* Method (for r=1).

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	$\bar{R}_{S_p}$	E_p	$F_p.T_s^{(e)}$
240	Ø	2.0	.230133608E-01	.168859959E-03	173.840	173.840	1	2.219	1.989	1	1
	Ø,1				89.050	178.100	1.952	4.332	3.884	0.976	1.905
	Ø,1,2				60.210	180.630	2.887	6.407	5.744	0.962	2.779
	Ø,1,2,3				45.580	182.320	3.814	8.463	7.587	0.953	3.637
480	Ø	2.0	.602386892E-02	.600218773E-04	346.080	346.080	1	2.184	1.981	1	1
	Ø,1				176.790	353.580	1.958	4.275	3.878	0.979	1.916
	Ø,1,2				118.830	356.490	2.912	6.360	5.770	0.971	2.827
	Ø,1,2,3				89.640	358.560	3.861	8.431	7.649	0.965	3.726
960	Ø	2.0	.590710714E-02	.588297844E-04	687.170	687.170	1	2.180	1.985	1	1
	Ø,1				350.930	701.860	1.958	4.269	3.888	0.979	1.917
	Ø,1,2				235.410	706.230	2.919	6.364	5.796	0.973	2.840
	Ø,1,2,3				177.970	711.880	3.861	8.418	7.666	0.965	3.727
1920	Ø	2.0	.529513881E-02	.527501106E-04	1374.860	1374.860	1	2.173	1.990	1	1
	Ø,1				697.480	1394.960	1.971	4.284	3.922	0.986	1.943
	Ø,1,2				469.710	1409.130	2.927	6.361	5.824	0.976	2.856
	Ø,1,2,3				354.100	1416.400	3.883	8.438	7.726	0.971	3.769
TIME-STEPS:			60								

Table IV.B.3.5-t4: Experimental Results and Performance Measurements of the Parallel Algorithm for the *(D).A.G.E.* Method (for r=2) on the *'NEPTUNE'* Prototype System.

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	$\bar{R}_{S_p}$	E_p	$F_p.T_s^{(e)}$
240	Ø	2.0	.438848920E-01	.329077244E-03	343.978	343.978	1	2.228	1.991	1	1
	Ø,1				175.798	351.596	1.957	4.359	3.896	0.978	1.914
	Ø,1,2				119.243	357.729	2.885	6.426	5.744	0.962	2.774
	Ø,1,2,3				90.163	360.652	3.815	8.499	7.597	0.954	3.639
480	Ø	2.0	.117268600E-01	.117063522E-03	679.780	679.780	1	2.207	1.998	1	1
	Ø,1				347.790	695.580	1.955	4.313	3.906	0.977	1.910
	Ø,1,2				234.200	702.600	2.903	6.405	5.800	0.968	2.808
	Ø,1,2,3				176.920	707.680	3.842	8.479	7.678	0.961	3.691
960	Ø	2.0	.961450860E-02	.961422920E-04	1353.520	1353.520	1	2.198	2.001	1	1
	Ø,1				691.470	1382.940	1.957	4.302	3.917	0.979	1.916
	Ø,1,2				464.180	1392.540	2.916	6.408	5.834	0.972	2.834
	Ø,1,2,3				350.020	1400.080	3.867	8.498	7.737	0.967	3.738
1920	Ø	2.0	.908853114E-02	.907778740E-04	2717.820	2717.820	1	2.183	1.998	1	1
	Ø,1				1382.310	2764.620	1.966	4.293	3.927	0.983	1.933
	Ø,1,2				927.250	2781.750	2.931	6.400	5.855	0.977	2.864
	Ø,1,2,3				695.850	2783.400	3.906	8.528	7.802	0.976	3.814
TIME-STEPS:			120								

Table IV.B.3.5-t4(cont.d.): Experimental Results and Performance Measurements of the Parallel Algorithm for the *(D).A.G.E.* Method (for r=2) on the *'NEPTUNE'* Prototype System.

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$T_c^{(e)}$ (secs)	C_p	S_p	R_{S_p}	$\bar{R}_{S_p}$	E_p	$F_p.T_s^{(e)}$
240	Ø	4.0	.448943786E-01	.336647034E-03	174.108	174.108	1	4.401	3.934	1	1
	Ø,1				89.375	178.750	1.948	8.574	7.664	0.974	1.897
	Ø,1,2				60.315	180.945	2.887	12.704	11.356	0.962	2.778
	Ø,1,2,3				45.648	182.592	3.814	16.786	15.005	0.954	3.637
480	Ø	4.0	.631995499E-02	.631213188E-04	344.220	344.220	1	4.358	3.946	1	1
	Ø,1				176.350	352.700	1.952	8.507	7.703	0.976	1.905
	Ø,1,2				118.780	356.340	2.898	12.630	11.436	0.966	2.799
	Ø,1,2,3				89.700	358.800	3.837	16.724	15.144	0.959	3.682
960	Ø	4.0	.657623261E-02	.654459000E-04	686.290	686.290	1	4.334	3.946	1	1
	Ø,1				352.480	704.960	1.947	8.439	7.683	0.974	1.895
	Ø,1,2				236.310	708.930	2.904	12.588	11.460	0.968	2.811
	Ø,1,2,3				178.710	714.840	3.840	16.649	15.154	0.960	3.687
1920	Ø	4.0	.512412935E-02	.507831573E-04	1378.870	1378.870	1	4.304	3.937	1	1
	Ø,1				703.590	1407.180	1.960	8.434	7.716	0.980	1.920
	Ø,1,2				472.130	1416.390	2.921	12.569	11.499	0.974	2.843
	Ø,1,2,3				354.480	1417.920	3.890	16.740	15.315	0.972	3.783
TIME-STEPS:			60								

Table IV.B.3.5-t5: Experimental Results and Performance Measurements of the Parallel Algorithm for the *(D).A.G.E.* Method (for r=4) on the *'NEPTUNE'* Prototype System.

G_S	P_H	PROCESSORS (p)		$Ac_{(i,j)}$	$Tc^{(t)}_{(i,j)}$ (secs)	I_{cl}	Ac_p	$Tc^{(t)}_p$ (secs)	$N_{p(p)}$	$I_{cl(//)}$	SHARED DATA		PARALLEL PATH		
		N_t	S_p								$R_{a(s)}$	$o^{(t)}_{st(s)}$	$R_{a(//)}$	$o^{(t)}_{st(//)}$	
240×60	3	$p \leq 2N_{B.P.}$ $[p \mid N_{B.P.}]$	$O(p)$	60 flops	0.023	$\frac{120}{p}$	$\frac{7.2\times10^3}{p}$ flops	$\frac{2.772}{p}$	1	1	1:60 flops	0.003%	1:$\frac{7.2\times10^3}{p}$ flops	0.043p%	***
	4	$p \leq N_{I.P.}$ $[p \mid N_{I.P.}]$	$O(p)$	56 flops	0.022	$\frac{240}{p}$	$\frac{13.44\times10^3}{p}$ flops	$\frac{5.174}{p}$	1	1	1:56 flops	0.004%	1:$\frac{13.44\times10^3}{p}$ flops	0.023p%	
	5	$p \leq N_{I.P.}$ $[p \mid N_{I.P.}]$	$O(p)$	49 flops	0.019	$\frac{120}{p}$	$\frac{5.88\times10^3}{p}$ flops	$\frac{2.264}{p}$	4	15	1:4 flops	0.049%	1:$\frac{5.88\times10^3}{p}$ flops	0.053p%	

Table IV.B.3.5-t6: A Program Dependent Performance Analysis of the Parallel Algorithm for the *(D).A.G.E.* Method (for r=4).

***	LIMITS TO PERFORMANCE			$Id^{(t)}_t$	W_{st} (secs)
	$S_{d(r)}$	$S'_{d(r)}$	$S_{h(r)}$		
	m_p=30,596	m_p=24,509	$m_p=\frac{2,310}{p}$	~1.5%	0.005
	m_p=28,556	m_p=22,875	$m_p=\frac{4,312}{p}$		0.005
	m_p=2,039	m_p=1,633	$m_p=\frac{1,886}{p}$		0.353

G_S	$T^{(e)}_s$ (secs)	S_p			$\sum_{\iota=1}^{p} c_{y_\iota}$	t_{cy} (μsecs)	$Id^{(e)}_t$	t_b (μsecs)	PARALLEL PATH		$T^{(e)}_s$ (secs)	PARALLEL CONTROL	$T^{(e)}_s$ (secs)	SHARED DATA	W_{dc} (secs)
	[XPFCL]	ø,1	ø,1,2	ø,1,2,3					$o^{(e)}_{st(//)}$	$o^{(e)}_{cn(//)}$	[XPFCLN]	$o^{(e)}_{tl(//)}$	[XPFCLS]	$o^{(e)}_{tl(s)}$	
240×60	174.108	1.948	2.887	3.814	689	~10,800	~4.1%	~686	0.17%	0.26%	173.880	0.13%	173.680	0.12%	7.441

Table IV.B.3.5-t7: A System Dependent Performance Analysis of the Parallel Algorithm for the *(D).A.G.E.* Method (for r=4).

i) The smallest run-time $T_p^{(e)}$ to be utilized in formula *(IV.B.3.1:25)*, when r=1, was 180.170 secs, while, when r=4, was ~45.638 secs;

ii) the total number of wait cycles to be utilized in formula *(IV.B.3.1:27)*, when r=1, was ~2710, while, when r=4, was ~689, which implied average numbers of wait cycles per processor of ~677 and ~172, respectively;

iii) the average experimental timings of all cooperating processors for these cases were ~180.176 secs and ~45.644 secs, accordingly; and,

iv) the number of parallel paths run by each processor, considering the average of all cooperating processors but P_0, in correspondence with the grid ratio values were 243 and 63, respectively.

Finally, the times that the system was not used productively (W), again estimated through the formula *(IV.B.3.1:22)* by using the average experimental timings in *iii)*, accordingly, were ~35.771 secs and ~8.468 secs, while, similarly as before, good approximations to these total wasted times can be obtained from the *sum* of the respective wasted times statically and dynamically, given in the performance analysis *Tables*.

The same conclusive remarks made for the parallel algorithm for the *(S)AGE* method and the same real processing-to-access† ratio, for the *fifth phase* of the algorithm, apply for this case as well.

---

†*To the shared data resource, for each implementation cycle.*

IV.B.3.6: A '(Modified Double) Alternating Group Explicit' - *(MD)AGE* Method: Experimental Results And Performance Analysis On The '*NEPTUNE*' Prototype System

This final scheme is based on the formation of the *(D)AGE* scheme as it was outlined in *Figure (IV.B.3-f5,iv)*, but it is a modified periodic rotation of its *three* first time-levels only. In specific, this scheme is given by

$$\left.\begin{aligned}(I+r\hat{G}_{2,j})\underline{U}_{j+1} &= (I-r\hat{G}_{1,j})\underline{U}_j + \underline{b}_4\\ (I+r\hat{G}_{1,j})\underline{U}_{j+2} &= (I-r\hat{G}_{2,j})\underline{U}_{j+1} + \underline{b}_3\\ (I+r\hat{G}_{1,j})\underline{U}_{j+3} &= (I-r\hat{G}_{2,j})\underline{U}_{j+2} + \underline{b}_3\end{aligned}\right\} \quad ; \qquad (IV.B.3.6:1)$$

the *brick* diagram for this scheme is given in *Figure (IV.B.3.6-f1)*.

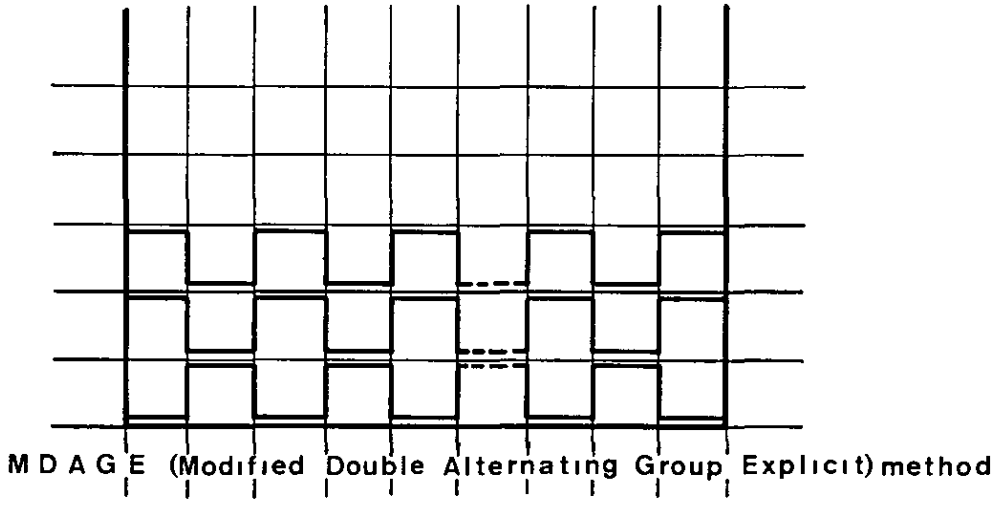


Figure IV.B.3.6-f1: The Representative Diagram of this Scheme.

The six-phased implemented program of this method is included in *Appendix C-IV* under the name *MB$5.MDAGEONI*[†].

Once more, there is a close correspondence between this implementation and that of the *(D)AGE* method, except that, in the *second phase*, the exact

[†] *It stands for Modified Double Alternating Group Explicit method for Odd Number of Intervals.*

theoretical values are computed at all the boundary and internal points, at the maximum time-level, using the exact solution formula *(IV.B.3.1:2)* and they are held in the non-shared real array *W*. In addition, in the *third phase*, the computation of the exact values at all the points on both boundaries starts from *zero* time-level up to the penultimate one using again the above formula. At the last parallel *fifth phase*, a similar *flag* is appropriately set as before to control the alternating asymmetric sequence between the *GEC* and *GEU* schemes, synthesizing the present step process. All the other program structuring specifications made for the previous parallel algorithm can equally well apply for the present one.

- Experimental Results

This method can be again, similar to the previous ones, proved to be unconditionally stable for all $r>0$, for the same region of the *open rectangle* $[0,1]\times[0,+\infty)$. The grid sizes experimented with and the corresponding results obtained, for the same as before values for r, are being presented in *Tables (IV.B.3.6-t1,t4,t5)*, accordingly.

A general comparison of these results with those of the previous unconditionally stable, for $r>0$, *GE* parallel algorithms, proves that the present algorithm behaves slightly worse in terms of the determining performance factors, i.e. the time-complexities achieved. Certainly, again, there is not any comparison with the *standard* algorithm from every aspect, while, in concern with the *GEU* and *GEC* methods, it is far better, in terms of the time-complexities, than the former and worse than the latter one. However, in a similar manner as before, the tremendous potential of the method, when exceeding the restrictive barrier of *1* for

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	E_p	$F_p.T_s^{(e)}$	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	E_p	$F_p.T_s^{(e)}$
240	Ø	1.0	.129027292E-01	.936388969E-04	177.393	177.393	1	1.101	1	1	.257103033E-01	.188648701E-03	350.420	350.420	1	1.101	1	1
	Ø,1				91.010	182.020	1.949	2.147	0.975	1.900			179.490	358.980	1.952	2.149	0.976	1.906
	Ø,1,2				61.668	185.004	2.877	3.168	0.959	2.758			121.660	364.980	2.880	3.171	0.960	2.765
	Ø,1,2,3				46.698	186.792	3.799	4.184	0.950	3.608			91.920	367.680	3.812	4.196	0.953	3.633
480	Ø	1.0	.541954488E-02	.541806221E-04	352.030	352.030	1	1.088	1	1	.103203431E-01	.103175640E-03	694.880	694.880	1	1.088	1	1
	Ø,1				180.160	360.320	1.954	2.125	0.977	1.909			355.630	711.260	1.954	2.125	0.977	1.909
	Ø,1,2				121.530	364.590	2.897	3.151	0.966	2.797			239.500	718.500	2.901	3.156	0.967	2.806
	Ø,1,2,3				91.910	367.640	3.830	4.166	0.958	3.668			180.650	722.600	3.847	4.184	0.962	3.699
960	Ø	1.0	.649359077E-02	.647902489E-04	703.020	703.020	1	1.081	1	1	.125266537E-01	.125050545E-03	1385.520	1385.520	1	1.081	1	1
	Ø,1				358.950	717.900	1.959	2.117	0.979	1.918			706.480	1412.960	1.961	2.120	0.981	1.923
	Ø,1,2				241.500	724.500	2.911	3.146	0.970	2.825			475.840	1427.520	2.912	3.148	0.971	2.826
	Ø,1,2,3				181.740	726.960	3.868	4.180	0.967	3.741			359.050	1436.200	3.859	4.172	0.965	3.723
1920	Ø	1.0	.662099198E-02	.661611557E-04	1408.700	1408.700	1	1.075	1	1	.124188215E-01	.124096870E-03	2774.870	2774.870	1	1.077	1	1
	Ø,1				714.820	1429.640	1.971	2.118	0.985	1.942			1405.300	2810.600	1.975	2.126	0.987	1.949
	Ø,1,2				480.960	1442.880	2.929	3.147	0.976	2.860			946.530	2839.590	2.932	3.157	0.977	2.865
	Ø,1,2,3				362.880	1451.520	3.882	4.172	0.970	3.767			712.590	2850.360	3.894	4.193	0.974	3.791
TIME-STEPS:			60								120							

Table IV.B.3.6-t1: Experimental Results and Performance Measurements of the Parallel Algorithm for the *(M.D).A.G.E.* Method (for r=1) on the *'NEPTUNE'* Prototype System.

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$T_c^{(e)}$ (secs)	C_p	S_p	R_{S_p}	E_p	$F_p.T_s^{(e)}$
240	Ø	1.0	.488925874E-01	.366628170E-03	694.593	694.593	1	1.103	1	1
	Ø,1				355.393	710.786	1.954	2.156	0.977	1.910
	Ø,1,2				240.938	722.814	2.883	3.180	0.961	2.770
	Ø,1,2,3				182.535	730.140	3.805	4.198	0.951	3.620
480	Ø	1.0	.209336840E-01	.209271908E-03	1378.700	1378.700	1	1.088	1	1
	Ø,1				703.450	1406.900	1.960	2.133	0.980	1.921
	Ø,1,2				474.920	1424.760	2.903	3.159	0.968	2.809
	Ø,1,2,3				359.150	1436.600	3.839	4.177	0.960	3.684
960	Ø	1.0	.218955763E-01	.218510628E-03	2746.810	2746.810	1	1.083	1	1
	Ø,1				1402.350	2804.700	1.959	2.121	0.979	1.918
	Ø,1,2				943.120	2829.360	2.912	3.154	0.971	2.827
	Ø,1,2,3				710.280	2841.120	3.867	4.188	0.967	3.739
1920	Ø	1.0	.232282057E-01	.232040882E-03	5518.160	5518.160	1	1.075	1	1
	Ø,1				2798.840	5597.680	1.972	2.120	0.986	1.944
	Ø,1,2				1882.650	5647.950	2.931	3.152	0.977	2.864
	Ø,1,2,3				1418.900	5675.600	3.889	4.182	0.972	3.781
TIME-STEPS:			240							

Table IV.B.3.6-t1(cont.d.): Experimental Results and Performance Measurements of the Parallel Algorithm for the *(M.D).A.G.E.* Method (for r=1) on the *'NEPTUNE'* Prototype System.

the grid ratio, establishes it far superior in comparison with the basic methods above. In fact, the best Speed-ups and Efficiencies achieved, for the particular case that r=1, were for the grid of size (1920×120), for every combination of cooperating processors in the system.

In accordance, the R_{S_p} and $\bar{R}_{S_p}$, parameters of the performance model exhibit Speed-ups of $>O(r.p)$ and $O(r.p)$, respectively; in specific, when r=4, we achieved a maximum, *Relative* to the *standard* algorithm, *acceleration* of 16.560 times, while the *Reference internal acceleration* exhibited an even greater maximum, than those of the *(S)AGE* and *(D)AGE* methods, of 15.399 times, both figures achieved in a total system utilization.

Finally, from the pure numerical point of view, the accuracy of this method generally increases along with the grid ratio r, the maximum absolute error, when r=4, achieving its smallest value when the grid size becomes maximum.

- Performance Analysis

In a corresponding manner as for the previous *GE* algorithms, *Tables (IV.B.3.6-t2,t3,t6,t7)* respectively present the program and system dependent performance analyses results for the extreme cases, in terms of the grid ratio, again.

The complementary information, essential for the estimation of the experiment dependent parameters in these *Tables*, similarly obtained from the shared array *ITIME* and following once more the same procedure as before, in a maximum system utilization, was as follows:

i) The smallest run-time $T_p^{(e)}$ to be utilized in formula *(IV.B.3.1:25)*, when r=1, was 182.525 secs, while, when r=4, was 46.500 secs;

G_S	P_H	PROCESSORS (p)		$Ac_{(i,j)}$	$Tc^{(t)}_{(i,j)}$ (secs)	I_{cl}	Ac_p	$Tc^{(t)}_p$ (secs)	$N_{p(p)}$	$C_{(//)}$	SHARED DATA		PARALLEL PATH	
		N_t	S_p								$R_{a(s)}$	$O^{(t)}_{st(s)}$	$R_{a(//)}$	$O^{(t)}_{st(//)}$
240×240	3	$p \lesssim 2N_{B.P.}$ $[p \mid N_{B.P.}]$	$\emptyset(p)$	60 flops	0.023	$\frac{480}{p}$	$\frac{28.8\times10^3}{p}$ flops	$\frac{11.088}{p}$	1	1	1:60 flops	0.003%	1:$\frac{28.8\times10^3}{p}$ flops	0.011p%
	4	$p \lesssim N_{I.P.}$ $[p \mid N_{I.P.}]$	$\emptyset(p)$	56 flops	0.022	$\frac{240}{p}$	$\frac{13.44\times10^3}{p}$ flops	$\frac{5.174}{p}$	1	1	1:56 flops	0.004%	1:$\frac{13.44\times10^3}{p}$ flops	0.023p%
	5	$p \lesssim N_{I.P.}$ $[p \mid N_{I.P.}]$	$\emptyset(p)$	49 flops	0.019	$\frac{120}{p}$	$\frac{5.88\times10^3}{p}$ flops	$\frac{2.264}{p}$	3	80	1:4 flops	0.049%	1:$\frac{5.88\times10^3}{p}$ flops	0.053p%

Table IV.B.3.6-t2: A Program Dependent Performance Analysis of the Parallel Algorithm for the *(M.D).A.G.E.* Method (for r=1).

LIMITS TO PERFORMANCE			$Id^{(t)}_t$	W_{st} (secs)
$S_{d(r)}$	$S'_{d(r)}$	$S_{h(r)}$		
n_p=30,596	m_p=24,509	$m_p=\frac{9,240}{p}$	~1.5%	0.005
n_p=28,556	m_p=22,875	$m_p=\frac{4,312}{p}$		0.005
n_p=2,039	m_p=1,633	$m_p=\frac{1,886}{p}$		1.413

G_S	$T^{(e)}_s$ (secs)	S_p			$\sum_{i=1}^{p} c_{y_i}$	t_{cy} (μsecs)	$Id^{(e)}_t$	t_b (μsecs)	PARALLEL PATH		$T^{(e)}_s$ (secs)	PARALLEL CONTROL	$T^{(e)}_s$ (secs)	SHARED DATA	W_{dc} (secs)
	[XPFCL]	∅,1	∅,1,2	∅,1,2,3					$O^{(e)}_{st(//)}$	$O^{(e)}_{cn(//)}$	[XPFCLN]	$O^{(e)}_{tl(//)}$	[XPFCLS]	$O^{(e)}_{tl(s)}$	
240×240	694.593	1.954	2.883	3.805	2673	~10,800	~4%	~686	0.16%	0.25%	693.986	0.09%	693.503	0.07%	28.868

Table IV.B.3.6-t3: A System Dependent Performance Analysis of the Parallel Algorithm for the *(M.D).A.G.E.* Method (for r=1).

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$Tc^{(e)}$ (secs)	C_p	S_p	R_{S_p}	$\bar{R}_{S_p}$	E_p	$F_p.T_s^{(e)}$
240	Ø	2.0	.239637904E-01	.175833702E-03	177.230	177.230	1	2.176	1.977	1	1
	Ø,1				90.780	181.560	1.952	4.249	3.860	0.976	1.906
	Ø,1,2				61.510	184.530	2.881	6.271	5.697	0.960	2.767
	Ø,1,2,3				46.490	185.960	3.812	8.297	7.538	0.953	3.633
480	Ø	2.0	.602386892E-02	.600218773E-04	349.950	349.950	1	2.160	1.986	1	1
	Ø,1				178.970	357.940	1.955	4.223	3.883	0.978	1.912
	Ø,1,2				120.650	361.950	2.901	6.264	5.759	0.967	2.804
	Ø,1,2,3				91.190	364.760	3.838	8.288	7.620	0.959	3.682
960	Ø	2.0	.676294789E-02	.673532486E-04	695.770	695.770	1	2.153	1.991	1	1
	Ø,1				355.140	710.280	1.959	4.218	3.901	0.980	1.919
	Ø,1,2				239.040	717.120	2.911	6.267	5.796	0.970	2.824
	Ø,1,2,3				180.130	720.520	3.863	8.317	7.692	0.966	3.730
1920	Ø	2.0	.533331186E-02	.532269478E-04	1396.850	1396.850	1	2.139	1.987	1	1
	Ø,1				709.520	1419.040	1.969	4.211	3.911	0.984	1.938
	Ø,1,2				477.180	1431.540	2.927	6.261	5.815	0.976	2.856
	Ø,1,2,3				360.090	1440.360	3.879	8.298	7.706	0.970	3.762
TIME-STEPS:			60								

Table IV.B.3.6-t4: Experimental Results and Performance Measurements of the Parallel Algorithm for the *(M.D).A.G.E.* Method (for r=2) on the *'NEPTUNE'* Prototype System.

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$T_c^{(e)}$ (secs)	C_p	S_p	R_{S_p}	$\bar{R}_{S_p}$	E_p	$F_p.T_s^{(e)}$
240	Ø	2.0	.458005331E-01	.343441963E-03	348.218	348.218	1	2.201	1.995	1	1
	Ø,1				178.475	356.950	1.951	4.293	3.892	0.976	1.903
	Ø,1,2				120.765	362.295	2.883	6.345	5.752	0.961	2.771
	Ø,1,2,3				91.488	365.952	3.806	8.376	7.592	0.952	3.622
480	Ø	2.0	.124581568E-01	.124216080E-03	691.640	691.640	1	2.169	1.993	1	1
	Ø,1				352.240	704.480	1.964	4.259	3.914	0.982	1.928
	Ø,1,2				237.510	712.530	2.912	6.316	5.805	0.971	2.827
	Ø,1,2,3				179.270	717.080	3.858	8.368	7.691	0.965	3.721
960	Ø	2.0	.110956579E-01	.110507011E-03	1370.370	1370.370	1	2.171	2.004	1	1
	Ø,1				702.800	1405.600	1.950	4.233	3.908	0.975	1.901
	Ø,1,2				471.130	1413.390	2.909	6.314	5.830	0.970	2.820
	Ø,1,2,3				355.740	1422.960	3.852	8.362	7.721	0.963	3.710
1920	Ø	2.0	.942506269E-02	.938773155E-04	2753.210	2753.210	1	2.155	2.004	1	1
	Ø,1				1397.330	2794.660	1.970	4.247	3.949	0.985	1.941
	Ø,1,2				939.420	2818.260	2.931	6.317	5.874	0.977	2.863
	Ø,1,2,3				708.180	2832.720	3.888	8.379	7.792	0.972	3.779
TIME-STEPS:			120								

Table IV.B.3.6-t4(cont.d.): Experimental Results and Performance Measurements of the Parallel Algorithm for the *(M.D).A.G.E.* Method (for r=2) on the *'NEPTUNE'* Prototype System.

$N_{I.P.}$	N_{PROCS}	r	$M_{P.E.}$	$M_{A.E.}$	$T_c^{(e)}$ (secs)	C_p	S_p	R_{S_p}	$\bar{R}_{S_p}$	E_p	$F_p.T_s^{(e)}$
240	Ø	4.0	.461820737E-01	.346302986E-03	176.988	176.988	1	4.329	3.925	1	1
	Ø,1				90.363	180.726	1.959	8.480	7.687	0.979	1.918
	Ø,1,2				61.405	184.215	2.882	12.479	11.312	0.961	2.769
	Ø,1,2,3				46.510	186.040	3.805	16.475	14.934	0.951	3.620
480	Ø	4.0	.650495663E-02	.649690628E-04	350.370	350.370	1	4.282	3.935	1	1
	Ø,1				179.020	358.040	1.957	8.380	7.701	0.979	1.915
	Ø,1,2				120.500	361.500	2.908	12.449	11.441	0.969	2.818
	Ø,1,2,3				91.010	364.040	3.850	16.483	15.149	0.962	3.705
960	Ø	4.0	.751056150E-02	.747442245E-04	697.800	697.800	1	4.263	3.936	1	1
	Ø,1				356.080	712.160	1.960	8.354	7.714	0.980	1.920
	Ø,1,2				238.840	716.520	2.922	12.454	11.501	0.974	2.845
	Ø,1,2,3				180.240	720.960	3.872	16.504	15.240	0.968	3.747
1920	Ø	4.0	.554303825E-02	.549554825E-04	1398.520	1398.520	1	4.243	3.946	1	1
	Ø,1				710.590	1421.180	1.968	8.351	7.766	0.984	1.937
	Ø,1,2				478.310	1434.930	2.924	12.406	11.537	0.975	2.850
	Ø,1,2,3				358.350	1433.400	3.903	16.560	15.399	0.976	3.808
TIME-STEPS:			60								

Table IV.B.3.6-t5: Experimental Results and Performance Measurements of the Parallel Algorithm for the *(M.D).A.G.E.* Method (for r=4) on the *'NEPTUNE'* Prototype System.

G_S	P_H	PROCESSORS (p)		$Ac_{(i,j)}$	$Tc^{(t)}_{(i,j)}$ (secs)	I_{cl}	Ac_p	$Tc^{(t)}_p$ (secs)	$N_{p(p)}$	$I\alpha(//)$	SHARED DATA		PARALLEL PATH		
		N_t	S_p								$R_{a(s)}$	$O^{(t)}_{st(s)}$	$R_{a(//)}$	$O^{(t)}_{st(//)}$	
240×60	3	$p \leq 2N_{B.P.}$ $[p \mid N_{B.P.}]$	$\emptyset(p)$	60 flops	0.023	$\frac{120}{p}$	$\frac{7.2\times10^3}{p}$ flops	$\frac{2.772}{p}$	1	1	1:60 flops	0.003%	1:$\frac{7.2\times10^3}{p}$ flops	0.043p%	•••
	4	$p \leq N_{I.P.}$ $[p \mid N_{I.P.}]$	$\emptyset(p)$	56 flops	0.022	$\frac{240}{p}$	$\frac{13.44\times10^3}{p}$ flops	$\frac{5.174}{p}$	1	1	1:56 flops	0.004%	1:$\frac{13.44\times10^3}{p}$ flops	0.023p%	
	5	$p \leq N_{I.P.}$ $[p \mid N_{I.P.}]$	$\emptyset(p)$	49 flops	0.019	$\frac{120}{p}$	$\frac{5.88\times10^3}{p}$ flops	$\frac{2.264}{p}$	3	20	1:4 flops	0.049%	1:$\frac{5.88\times10^3}{p}$ flops	0.053p%	

Table IV.B.3.6-t6: A Program Dependent Performance Analysis of the Parallel Algorithm for the *(M.D).A.G.E.* Method (for r=4).

LIMITS TO PERFORMANCE			$Id^{(t)}_t$	W_{st} (secs)
$S_{d(r)}$	$S'_{d(r)}$	$S_{h(r)}$		
m_p=30,596	m_p=24,509	$m_p=\frac{2,310}{p}$	~1.5%	0.005
m_p=28,556	m_p=22,875	$m_p=\frac{4,312}{p}$		0.005
m_p=2,039	m_p=1,633	$m_p=\frac{1,886}{p}$		0.353

G_S	$T^{(e)}_s$ (secs)	S_p			$\sum_{i=1}^{p} c_{y_i}$	t_{cy} (μsecs)	$Id^{(e)}_t$	t_b (μsecs)	PARALLEL PATH		$T^{(e)}_s$ (secs)	PARALLEL CONTROL	$T^{(e)}_s$ (secs)	SHARED DATA	W_{dc} (secs)
	[XPFCL]	∅,1	∅,1,2	∅,1,2,3					$O^{(e)}_{st(//)}$	$O^{(e)}_{cn(//)}$	[XPFCLN]	$O^{(e)}_{tl(//)}$	[XPFCLS]	$O^{(e)}_{tl(s)}$	
240×60	176.988	1.959	2.882	3.805	718	~10,800	~4.2%	~686	0.16%	0.27%	176.496	0.28%	176.200	0.17%	7.754

Table IV.B.3.6-t7: A System Dependent Performance Analysis of the Parallel Algorithm for the *(M.D).A.G.E.* Method (for r=4).

ii) the total number of wait cycles again for formula *(IV.B.3.1:27)*, when r=1, was ~2673, while, when r=4, was ~718, which implied average numbers of wait cycles per processor of ~668 and ~180, respectively;

iii) the average experimental timings of all cooperating processors for these cases were ~182.531 secs and 46.505 secs, accordingly; and,

iv) the number of parallel paths run by each processor, considering the average of all cooperating processors but P_0, in correspondence with the grid ratio values were 243 and 63, respectively.

To conclude, the times that the system was not used productively (W), estimated through formula *(IV.B.3.1:22)* by using the average experimental timings in *iii)*, accordingly, were ~35.531 secs and 9.032 secs, figures which can be approximated by the *sum* of the wasted times statically and dynamically, appearing in the appropriate *Tables* previously given. The real processing-to-access[†] ratio, for the *fifth phase* of the algorithm, has retained the same value as for the previous parallel algorithms presented herein.

IV.B.3.7: Indicative Experimental Results And Performance Measurements Of The *GEU* Method On SIMD And Pipelined Vector Computers

Our study of the potential and suitability of the *GE* methods for parallel computers will be completed by briefly reporting some results obtained from the implementation of the *GEU* method[‡], again for Burgers' equation *(IV.B.2:2)* with *(IV.B.3.1:2)* as the exact solution formula, on

---

[†] *To the shared data resource, for each implementation cycle.*

[‡] *See Evans, et al [EVAN83a].*

two different types of computers: The *ICL DAP* and the *CRAY-1S*, the particular characteristics of which have been discussed in the earlier surveying *Chapters*.

With respect to the *DAP* system, the different modes of operation available (i.e. *vector* and *matrix* modes) determine the subdivision of the considered interval in the x direction for a particular problem; however, note that the vector mode is of order of 5 to 6 times faster than the matrix mode for corresponding vector and matrix operations.

In fact for any *GE* scheme, if the interval in the x direction is divided into any number up to 64 or 4096 points, then the most effective operation mode for each instance would be the vector and matrix mode, respectively. However, since the grid points are grouped in pairs of two and the evaluation of each is different to that of the other, a fact opposing *DAP's* philosophy for identical operations, in the *vector* mode, for all 64 processors to be effectively utilized, the interval [0,1] in the x direction is divided into 128 sub-intervals; then, in accordance with the formula utilized for each grid point, two sets of 64 points are formed which can be evaluated in two separate sweeps. On the other hand, in the *matrix* mode, this division is increased to 8192 sub-intervals and two similar separate sets of 4096 grid points.

With respect to the *CRAY-1S* vector computer, it employs multiple pipelined functional units in lieu of processing elements. This computer usually consists of 8 or 16 separate memory banks and in theory up to 8 or 16 elements can be accessed simultaneously. The algorithms to be implemented on this system should, as much as possible, be expressed in a set of vector operations to be executed very fast. However, since most of the real time procedures cannot be fully vectorized, the performance

of such systems heavily depends upon the balance between the speed of the system for vector and sequential operations; therefore, this implies that vector machines should be provided with a reasonably fast sequential functional unit compatible with the speed of the vector units. In addition, and although it is tempting to think that their performance improves as the vector length of the vector operations increases[†], in real terms, increased parallelism (i.e. vector length) can only be achieved by incurring certain amounts of overhead costs, while to minimize memory access conflicts different vector elements should be stored in separate memory banks. A major advantage on this system compared with the *DAP* is that there are no constraints upon the sub-division of the considered interval on the x axis.

Note that, despite the very small values of k, due to the very fine subdivision of the interval [0,1] - up to a maximum of 8192 points, the results at each time-step were so accurate that there was no need for any iteration of the non-linear problem.

The following *Table (IV.B.3.7-t1)* illustrates the results of the experiments on both systems above, while the number of time-sweeps was 3000 for the parallel algorithm for the *GEU* method and 6000 for the *standard* algorithm. The accuracy obtained in both schemes was of order 0.1E-08.

The number of grid points was taken to be 4096, while the grid ratio values were chosen to be 1 and 1/2, respectively. The Speed-up shown for the *DAP* system was, consequently, obtained when approximately half of its processing power was effectively utilized, since to consider 8192

[†] *Since the starting up time associated with the functional unit will have less effect on the total time.*

grid points to achieve its full processing power can be proved unrealistic in practice. However, if the latter case occurs then the Speed-up can increase to about 15 when compared with the *ICL 2980* system.

DAP		*CRAY-1S*		
$Tc^{(e)}$ (secs)	$S_p^{\dagger}$	$Tc^{(e)}$ (secs)	$S_p^{\ddagger}$	
11.8	18.6	4.013	6.56	*Standard Explicit Method*
44.2	9.8	11.5	3.48	*GEU Method*

- †*Speed-up compared to the ICL 2980 (The DAP host)*
- ‡*Speed-up compared to the CRAY-1S without vectorization*

Table IV.B.3.7-t1: Experimental Results on the *'DAP'* and *'CRAY-1S'* Systems.

Hence, the irregularity and complexity between the computation of the two different formulae involved for each pair of grid points on *DAP* prove to be very uneconomical, since they reduce the processing power available.

To conclude, again similar remarks as before, for the unconditional stable for $r>0$ *GE* methods, can be made when choosing greater values for r. In specific, when $r=4$, only one quarter of the number of time-steps will be required to reach the same time-advance length, than when $r=1$, and consequently the timings will be reduced by a factor of 4, while the final accuracy more or less remains the same.

IV.B.4: Relative Performance Comparisons And Conclusive Remarks On The *GE* Methods

We shall conclude this *Chapter* by presenting a diagrammatical description of the parallel behaviour of all the *GE* schemes experimented with herein, in comparison with the Standard Explicit method. These diagrams apparently demonstrate the power and the merits of these new schemes, illustrating[†] our comparative remarks made when examining them earlier in this *Chapter*. Note that, all the occurring similarities amongst the program dependent performance analysis *Tables* are due to the fact that this analysis was only concerned with the *flops* of the algorithms, ignoring all integer operations, shared and local transfers, etc., as well as the rounding errors introduced in the arithmetic. The differences between the algorithms become apparent examining all the experimentally estimated parameters.

In particular for the diagrams, *Figures (IV.B.4-f1,f2)* correspondingly exhibit the experimental Time-complexities obtained on the *NEPTUNE* prototype system and the respective Speed-ups achieved. In *Figures (IV.B.4-f3,f4)* the Relative (or normalized) and Reference internal Speed-ups are depicted, while the Efficiency and the real Costs of all schemes are given in the last two *Figures (IV.B.4-f5,f6)*, respectively.

A comparison of the Algebraic-complexity of the *GE* schemes with the Standard Explicit and Crank-Nicolson methods is now necessary, which will prove that the *GE* method will be able to preserve the simplicity of the Explicit method.

In the following *Table (IV.B.4-t1)* the number of arithmetic operations involved to evaluate the solution at a point for all schemes

[†] *For those grid sizes for which the performance analyses were carried out.*

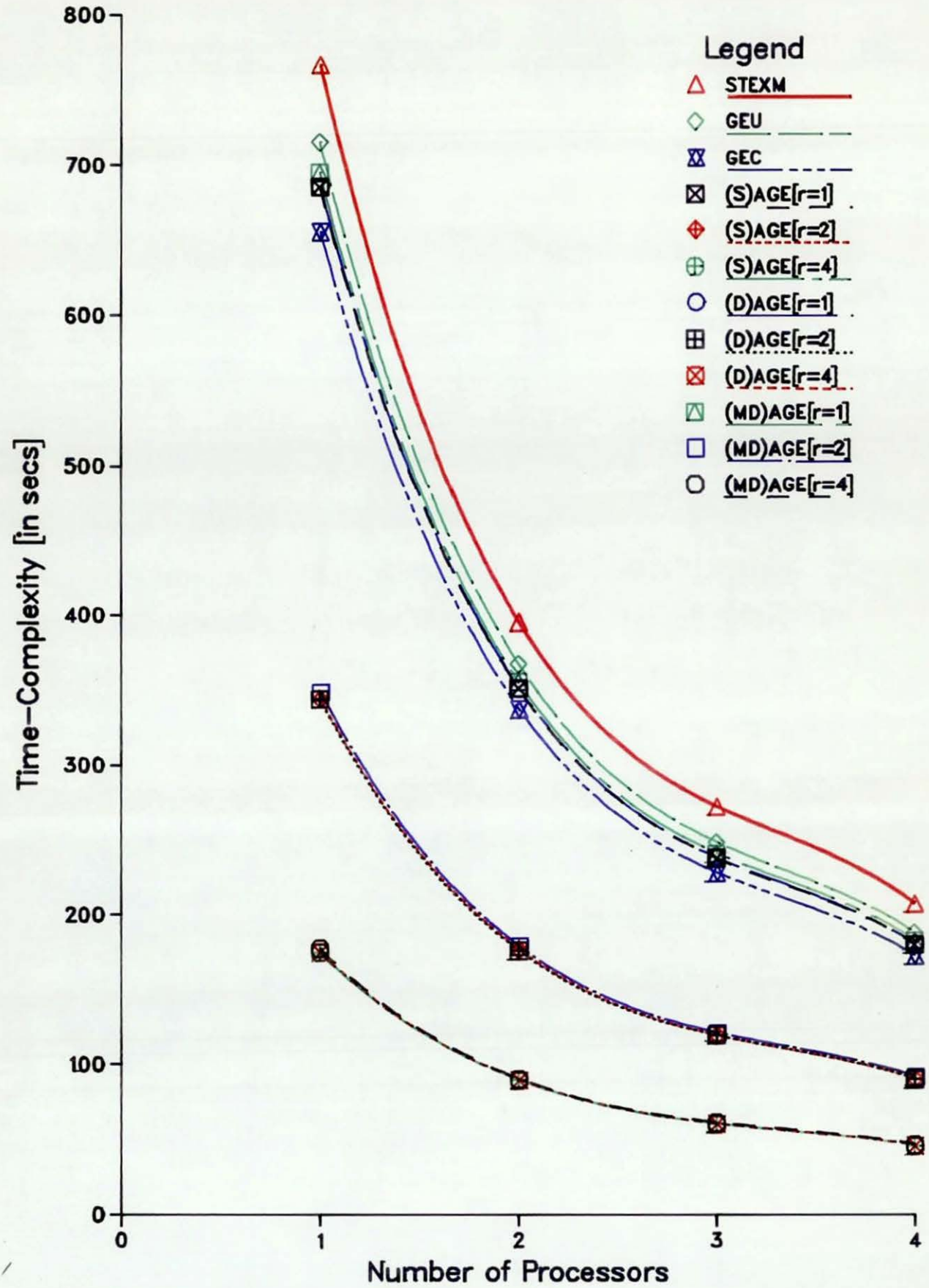


Figure IV.B.4-f1: The *Time-Complexity* of the Parallel Algorithms for the *GE* Schemes and the Standard Explicit Method.

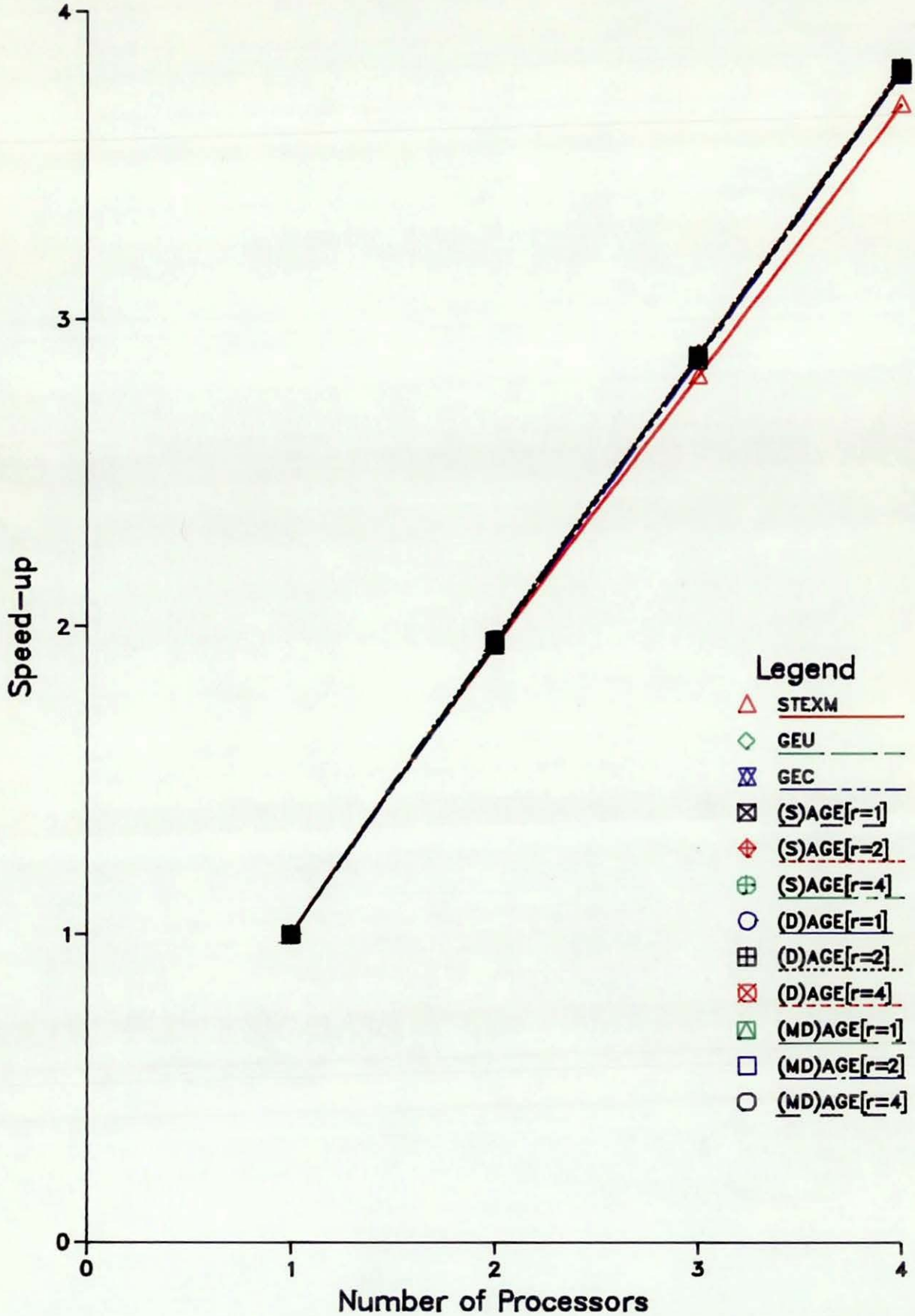


Figure IV.B.4-f2: The *Speed-ups* achieved by the Parallel Algorithms for the *GE* Schemes and the Standard Explicit Method.

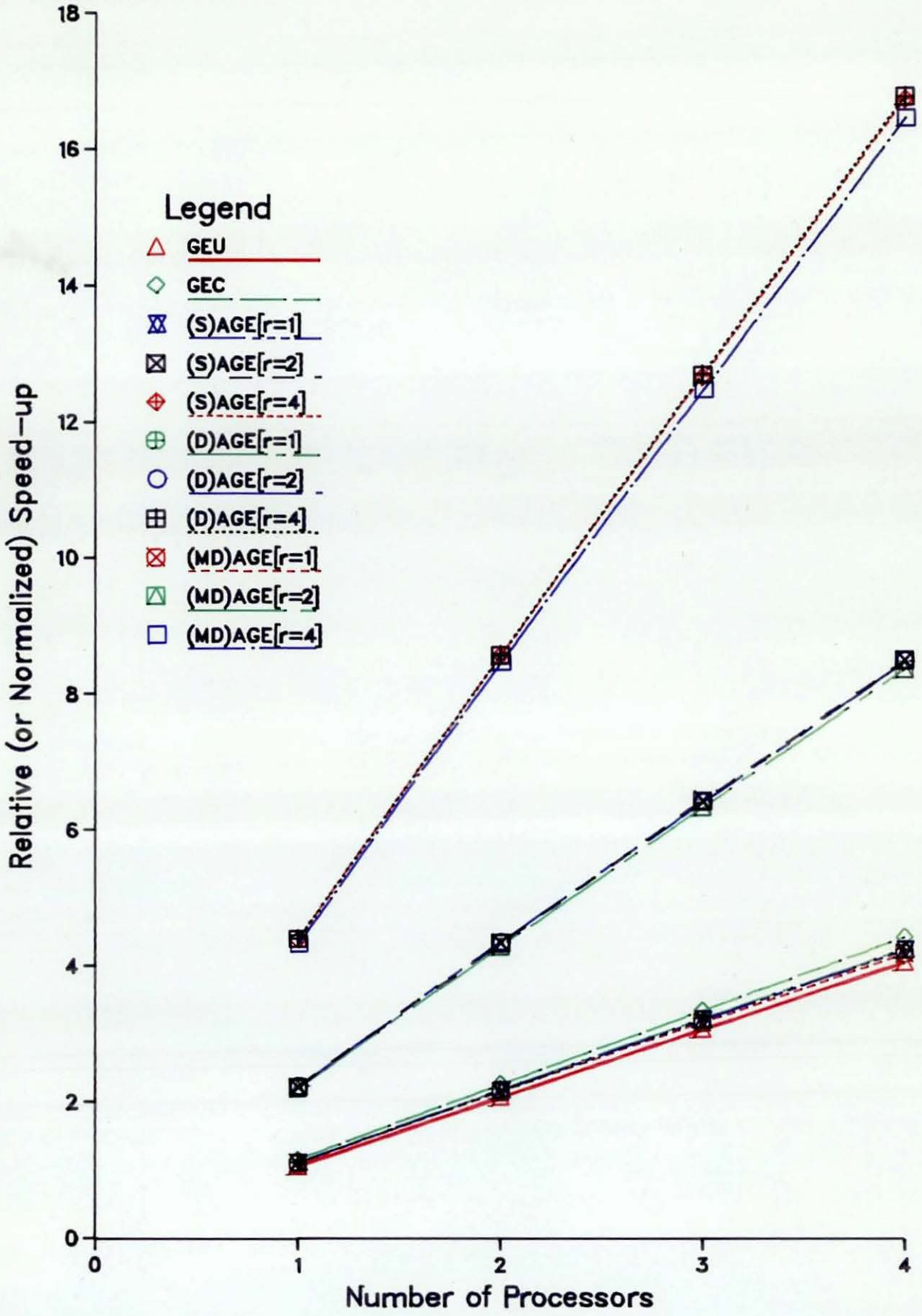


Figure IV.B.4-f3: The *Relative* (or *Normalized*) *Speed-ups* achieved by the Parallel Algorithms for the *GE* Schemes.

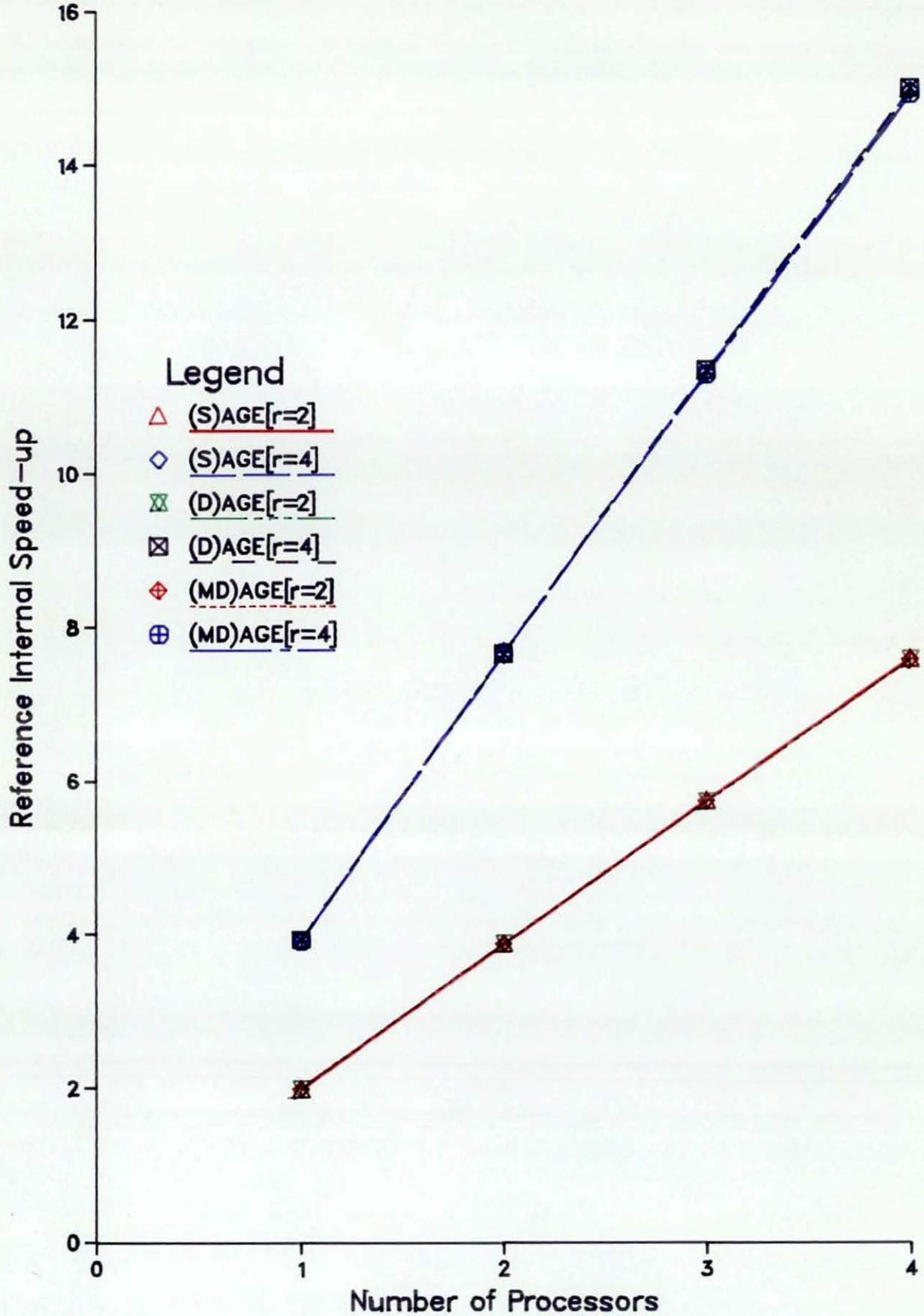


Figure IV.B.4-f4: The *Reference Internal Speed-ups* achieved by the Parallel Algorithms for the Unconditionally Stable (for all $r>0$) *GE* Schemes.

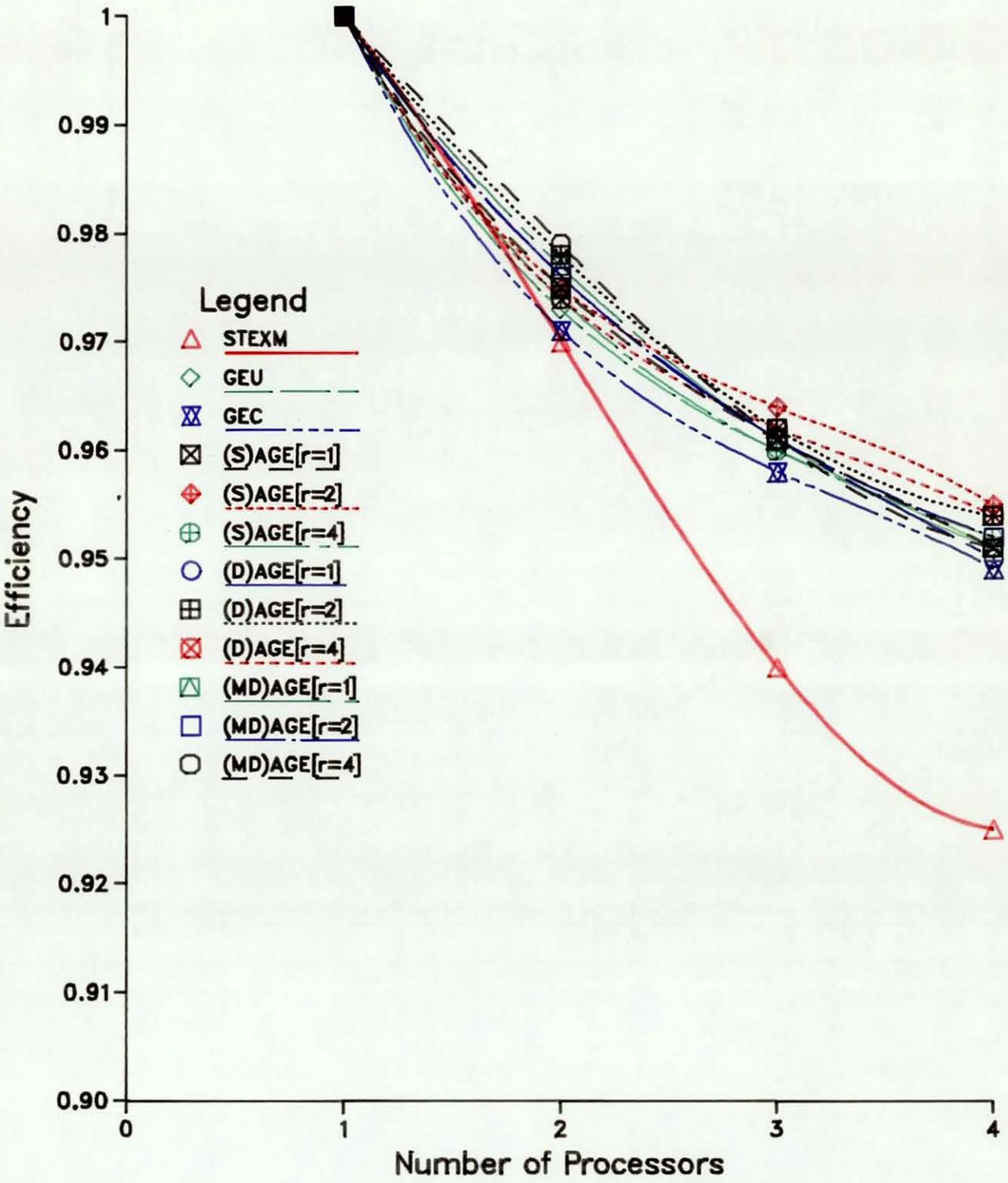


Figure IV.B.4-f5: The *Efficiency* achieved by the Parallel Algorithms for the *GE* Schemes and the Standard Explicit Method.

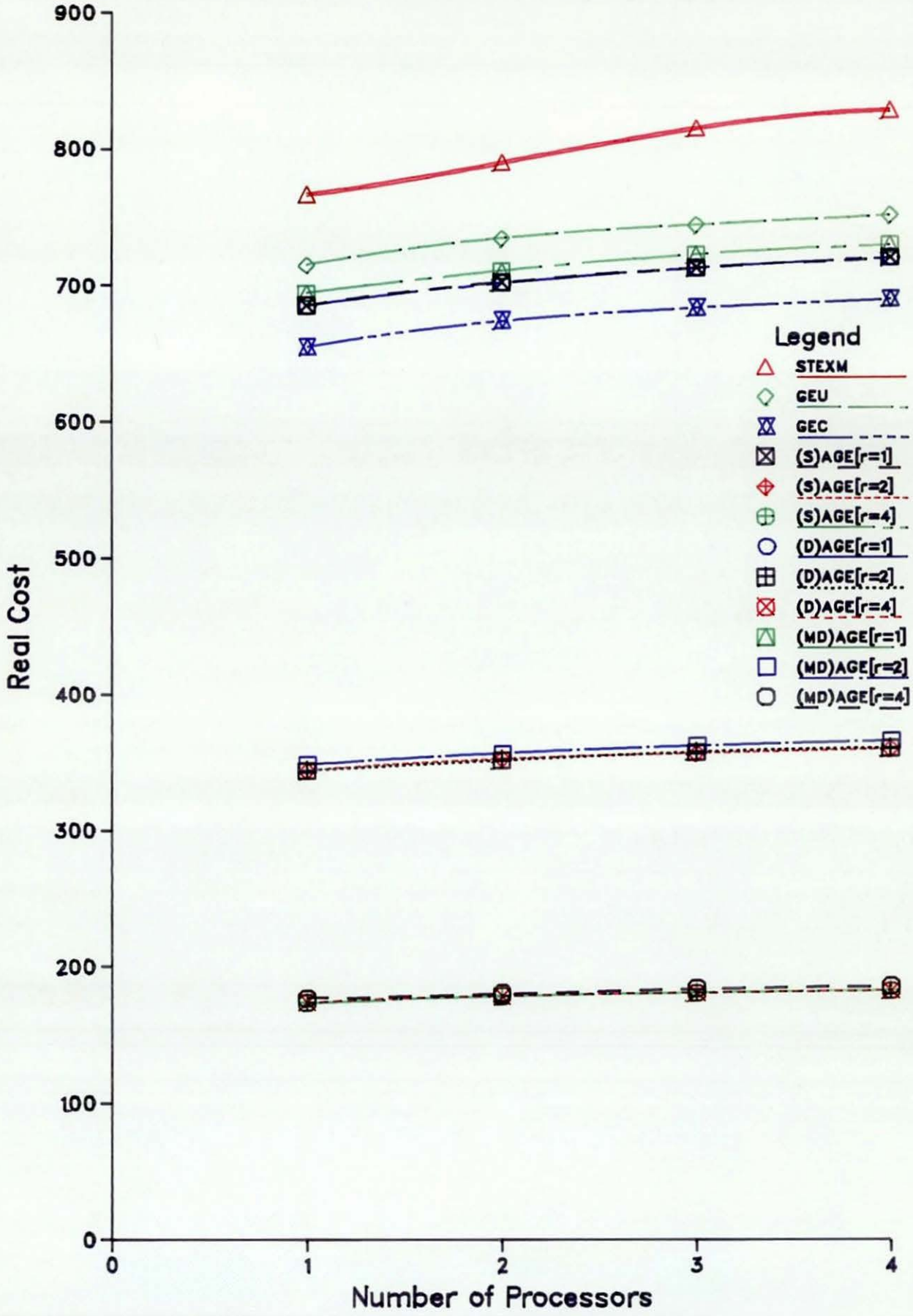


Figure IV.B.4-f6: The *Real Cost* of the Parallel Algorithms for the *GE* Schemes and the Standard Explicit Method.

in this class of methods, in comparison with the Standard Explicit and Crank-Nicolson methods, is given. For the latter *CN* method the figure given is the average from the number for solving the implicit system.

This *Table* shows that very much better stability characteristics are achieved over the Explicit method at the cost of some additional computational expense, i.e. approximately double, but this is still 30% less than that of the *CN* method. As far as the *storage* requirement is concerned, all the above methods are similar. Even more important, the explicitness of the new formulae has been retained and so are ideally suitable for parallel implementation.

OPERATION / METHOD	ADDITION	MULTIPLICATION	DIVISION
GE (Ordinary points)	3	4	1
GE (Ungrouped points)	2	2	2
CN (Average per point)	$\frac{(5n-1)}{n} \sim 5$	$\frac{(5n-1)}{n} \sim 5$	$\frac{(2n-1)}{n} \sim 2$
Explicit	2	2	0

Table (IV.B.4-t1): The Algebraic-Complexity of the *GE* Schemes in Comparison with the Standard Explicit and Crank-Nicolson Methods.

Finally, the concepts of the *GE* schemes can be, similarly, extended to the case of a two- and three-space dimensional problem (see Abdullah [*ABDU83*]). For the latter case we have groups of 8 grid points taken to form a *cube*, instead of the 4 points in a *plane* as for the two-dimensional problem. It is expected that the parallel behaviour of these schemes will improve as the number of dimensions increases.

N O T E S - N O T E S - N O T E S

N O T E S - N O T E S - N O T E S

N O T E S - N O T E S - N O T E S

N O T E S - N O T E S - N O T E S