Parallel applications and solutions in artificial intelligence and expert systems

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

Additional Information:

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

Metadata Record: https://dspace.lboro.ac.uk/2134/13692

Publisher: © K. R. Tout

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

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

AUTHOR/FILING TITLE

TOOT, K R

ACCESSION/COPY NO.

036000383

VOL. NO. CLASS MARK

2 NOV 1994

28 JUN 1993

LOAN COPY

20 JUN 1995

LOAN 3 WKS. + 3

28 JUN 1996

UNLESS RECALLED

21 JUN 1994

28 JUN 1996

(26 JUN 1996)

ACCOMPANYING DISK HELD AT ISSUE DESK

036000383 4
Parallel Applications and Solutions in Artificial Intelligence and Expert Systems

by

Kifah Raafat Tout

A Doctoral Thesis
Submitted in Partial Fulfilment of the Requirements
For the Award of Doctor of Philosophy
of Loughborough University of Technology

Supervisor: Professor D. J. Evans, Ph.D., D.Sc.,
Department of Computer Studies.

Declaration

I declare that I am responsible for the work submitted in this thesis, that the original work is my own except as specified in acknowledgements, and that neither the thesis nor the original work contained therein has been submitted to this or any other institution for a higher degree.

K. R. Tout.
Acknowledgements

The author wishes to express his gratitude to Professor D. J. Evans for the guidance, suggestions and assistance readily proffered throughout the course of the research and preparation of this thesis.

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

My father Professor Raafat Tout,

My lovely mother Khadije,

My brother Samer,

My sister Safaa and

My sister Amal her husband Tarek

and her lovely family

With Love

Kifah
Abstract

The work presented in this thesis focuses on the design and implementation of parallel algorithms for problem solving tasks principally in Rule-based Expert Systems and Artificial Intelligence (AI).

Rule-based Expert Systems are widely used in AI. Their use covers a wide variety of application areas. However, in most cases, these systems are computation intensive and run slowly. This increases the need for high-performance and real-time response.

Because of the convergence of parallelism in computer design and the wide spread use of expert system in industry, the design of Parallel Expert System has become of increasing importance. Parallel computation may prove useful in shortening the processing time of the expert systems.

Expert systems are being designed for both distributed (loosely-coupled) and shared-memory (tightly-coupled) multiprocessor machines. The work presented here is an attempt to focus on the issues involved in designing a rule-based expert system for a shared memory multiprocessor system (the Sequent Balance 8000).

Eight parallel Forward Chaining models and two parallel Backward Chaining models are implemented. These models are presented in Chapter 5 and 6, together with a study of their efficiency.
Contents

1 INTRODUCTION ......................... 1
   1.1 Expert Systems ........................................ 1
   1.2 Parallel Architectures .................................. 3
   1.3 Parallel Expert Systems .................................. 5

2 PARALLEL COMPUTER ARCHITECTURES .......... 7
   2.1 Introduction ........................................... 7
   2.2 Classification of Designs ................................. 9
       2.2.1 Flynn's Taxonomy .................................. 9
       2.2.2 Other Classifications ............................... 12
   2.3 Structural Classification of Computer Systems .......... 16
   2.4 Parallel Computer Structures ............................ 19
       2.4.1 Pipeline Computers .................................. 20
       2.4.2 Array Processors .................................... 23
       2.4.3 Multiprocessor Systems ............................. 30
           2.4.3.1 Interconnection Networks ...................... 31
           2.4.3.2 Functional Structures ......................... 38
   2.5 The Sequent Balance Multiprocessor System ........... 45
       2.5.1 System Hardware .................................... 45
       2.5.2 System Software .................................... 50
   2.6 Parallel Computer Architectures for Artificial Intelligence (AI) 52
       2.6.1 Characteristic of AI Computations ................. 52
       2.6.2 Language-Based AI Machines ....................... 54
       2.6.3 Knowledge-Based AI Machines ..................... 54
       2.6.4 Connectionist Machines ............................ 55
       2.6.5 Intelligent Interface Machines .................... 57
       2.6.6 The Columbia DADO Production System Machine .... 57

3 INTRODUCTION TO PARALLEL PROGRAMMING ...... 59
   3.1 Introduction ........................................... 59
   3.2 Parallelization ......................................... 62
       3.2.1 Implicit Parallelism ................................ 63
           3.2.1.1 Introduction .................................. 63
### 3.2. Program Dependence
- **3.2.1. Program Dependence**
- **3.2.2. Explicit Parallelism**

### 3.3. Elements of Parallel Programming for the Balance 8000
- **3.3.1. Programming Methods**
  - **3.3.1.1. Data Partitioning**
  - **3.3.1.2. Function Partitioning**
- **3.3.2. Process Creation and Termination**
- **3.3.3. Creation of Shared and Private Data**
- **3.3.4. Tasks Scheduling**
- **3.3.5. Tasks Synchronization and Mutual Exclusion**

### 3.4. Parallel Algorithms For Multiprocessors
- **3.4.1. Synchronized Parallel Algorithms**
- **3.4.2. Asynchronous Parallel Algorithms**
- **3.4.3. Performance of Parallel Algorithms**

### 4. INTRODUCTION TO EXPERT SYSTEMS

#### 4.1. A Brief History
#### 4.2. What is an Expert System
#### 4.3. Building an Expert System
#### 4.4. Main Structure of an Expert System
  - **4.4.1. The Knowledge Base**
  - **4.4.2. The Inference Engine**
  - **4.4.3. The User Interface**

#### 4.5. Knowledge Representation in Expert Systems
  - **4.5.1. Rules**
  - **4.5.2. Semantic Nets**
  - **4.5.3. Frames**

#### 4.6. Types of Tasks Carried out by Expert Systems
#### 4.7. Expert System Application Areas and Examples
#### 4.8. Rule-Based Expert Systems
  - **4.8.1. Introduction**
  - **4.8.2. Strategies for State Space Search**
    - **4.8.2.1. Forward-Chaining and Backward-Chaining Mechanisms**
    - **4.8.2.2. Depth-First and Breadth-First Search**
    - **4.8.2.3. AND/OR Tree**

#### 4.9. Background and Related Research
#### 4.10. Other Research Topics in Expert Systems

### 5. THE PARALLEL FORWARD CHAINING MODELS: DESIGN AND IMPLEMENTATION

#### 5.1. Introduction
#### 5.2. The Forward Chaining Process
CHAPTER I

Introduction
1.1 Expert Systems

The field of Expert systems research is still relatively new and there is considerable discussion among researchers as to the definition and direction of expert system work. An expert system is a program that behaves like an expert in some problem domain. Expert systems have to be capable of solving problems that require expert knowledge in some form. Therefore they are called knowledge-based systems. The process of building an expert system is often called knowledge engineering, and the expert-system builder is called a knowledge engineer.

All expert systems include at least three basic elements:

1. The knowledge base.
2. The inference engine.
3. The user interface.

The knowledge base represents what is known about a given subject at the present time. The inference engine knows how to actively use the knowledge in the base. The user interface facilitates communication between the user and the system; the inference engine and the user interface are often viewed as one module, usually called an expert system shell or simply a shell.

The three most widely used knowledge representation techniques, in current expert systems are:

1. The rules.
2. The semantic nets.
3. The frames.
Rule-based expert systems (also called production systems) are those systems that use a set of IF-THEN rules (also called productions) to present the domain knowledge. These systems consist of three main parts:

1. The **database**: which contains the known facts about the problem to be solved and the derived data or facts.
2. The **rule base**: that stores the rules used by the system. A rule has the following form:

   If $C_1 \& C_2 \& \ldots \& C_n$ THEN $A_1, A_2, \ldots, A_m$ where $C_{i=1,n}$ are the conditions of the rule and the $A_{i=1,m}$ are the actions or the conclusions of the rule.

3. **A control strategy or inference engine**: It controls the whole operation of the system.

   Rule-based expert systems use three reasoning mechanisms in their search for a solution, these are:

   a. The forward chaining (also called data-driven).
   b. The backward chaining (also called goal-driven).
   c. A mixture of both forward chaining and backward chaining.

   In forward chaining, the system chains forward from the conditions of the rules toward the conclusions in the search for a solution to the problem in hand. The rule's conditions are matched against the elements of the database in order to find all the applicable rules. A rule is applicable if all its conditions are satisfied by the content of the database. In contrary, a backward chaining system chains backward from the conclusions in the right-hand side of the rule towards its conditions part. Nowadays, expert systems are becoming more and more usable in a
great number of different areas, these include: Agriculture, Chemistry, Computer systems, Electronics, Engineering, Geology, Information management, Law, Manufacturing, Mathematics, Medicine, Meteorology, Military science, Physics, Process control, Space technology, Teaching and tutoring, Business, Cartography, Portfolio, Management, Statistics, Finance and economics, Scheduling, Banking, Regulation handling, Material and structure, etc . . .

Most of these applications fall into one or other of the following categories: interpretation, design, prediction, planning, monitoring, debugging, repair, instruction and control.

1.2 Parallel Architectures

Many significant scientific and technological problems require the use of large amounts of computing time. High performance computers are increasingly in demand in many scientific and engineering applications. The most challenging requirement has always been the attainment of high computing speeds. It is becoming increasingly difficult to design and build more powerful computers, even with the continuing advance in microelectronics technology. In order to solve these problems, the large-scale parallel computer has been developed. Parallel computers are those systems that emphasize parallel processing. The three main architectural configurations of parallel computers are:

i) Pipeline computers.

ii) Array Processors.

iii) Multiprocessor systems.
A pipelined computer performs overlapped computation to exploit temporal parallelism. An array processor uses multiple arithmetic logic units in a synchronized manner. Finally, a multiprocessor system achieves asynchronous parallelism through a set of interactive processors with shared resources (memories, database).

In addition to these three concepts, there are new approaches to parallel processing such as data flow, reduction and VLSI approaches. In data flow computers, an instruction is executed whenever its required operands become available (non-sequential operation). The reduction approach consists of carrying out instructions when results are needed for other calculations. Finally, the main idea of VLSI is to implement enormous numbers of devices on a chip. The rapid advent of VLSI technology, has created a new architectural horizon in implementing parallel algorithms directly in hardware. Multiprocessor systems may be classified into two main classes:

1. The tightly-coupled systems.
2. The loosely-coupled systems.

In loosely-coupled systems each processor has a set of input-output devices and a large local memory where it accesses most of the instructions and data. Processors communicate by exchanging messages through a message-transfer system. The communication topology of the associated message-transfer system is the determining factor of the degree of coupling in loosely-coupled systems.

Tightly-coupled systems can be defined as a large number of processors sharing a common parallel memory (the main memory), and controlled by a single operating
The processors communicate with each other through the shared main memory. Hence the rate at which data can communicate from one processor to the other is of the order of the bandwidth of the memory. A small local memory or high-speed buffer (cache) may exist in each processor. Cache memories are typically five to ten times faster than main memory. Contrary to loosely-coupled systems which are usually efficient when the interactions between tasks are minimal, tightly-coupled systems can tolerate a higher degree of interaction between tasks without significant deterioration in performance. In loosely-coupled systems the throughput may be too low for some applications that require fast response times. If high-speed or real-time processing is desired tightly-coupled systems may be used.

1.3 Parallel Expert Systems

Rule-based expert systems are widely used in Artificial Intelligence. Their use covers a wide variety of application areas. However, in most cases, these systems are computation intensive and run slowly. This increases the need for high-performance and real-time response.

Because of the convergence of parallelism in computer design and the widespread use of expert system in industry, the design of parallel expert system has become of increasing importance. Parallel computation may prove useful in shortening the processing time of the expert systems.

Expert Systems are being designed for both distributed (loosely-coupled) and shared memory (tightly-coupled) multiprocessor machines. The work presented in this thesis is an attempt to focus on the issues involved in designing a rule-based expert system for a shared memory multiprocessor system. The machine used is the
Sequent Balance 8000 a shared memory multiprocessor system. In Chapter 2 we give a general survey of parallel computer architectures and their classification. loosely-coupled and tightly-coupled multiprocessor architectures are presented, followed by a detailed description of the Sequent Balance.

Chapter 3 is a brief introduction to parallel programming and parallel algorithm design.

Chapter 4 introduces artificial intelligence and expert systems, and the techniques that are often used to implement expert systems. The most common structure of an expert system, and the techniques of reasoning used in the inference engine are described. Examples of existing expert systems, and application areas of expert systems conclude this chapter.

Chapter 5 examines the issues of the parallel implementation of the forward chaining technique in the context of a rule-based expert system and a shared memory parallel multiprocessor machine (Balance 8000). Eight parallel forward chaining models are presented, followed by a study of their efficiency.

In Chapter 6 the parallel implementation of backward-chaining techniques for rule-based expert systems, on the Sequent Balance 8000 are studied, analysed and discussed. Two parallel backward-chaining models are presented, followed by a study of their efficiency. A summary, conclusion and recommendations for further research in parallel implementation of backward-chaining system conclude this chapter.

Finally, a summary and general conclusions are the subject of the last chapter.

The thesis concludes with a list of references and a selection of parallel programs as Appendices.
CHAPTER II

Parallel Computer Architectures
2.1 Introduction

Many significant scientific and technological problems require the use of large amounts of computing time. High-performance computers are increasingly in demand in the areas of structural analysis, weather forecasting, petroleum exploration, fusion energy research, medical diagnosis, aerodynamics simulations, artificial intelligence, expert systems, industrial automation, remote sensing, military defense, genetic engineering, and socioeconomics, amongst many other scientific and engineering applications. In order to solve these problems, and achieve a reasonable computing time, the large-scale parallel computer has been developed.

In fact, until recently, the major share of this increase in computation speed has been derived from the development of faster electronic devices. Over the past four decades the computer industry has experienced four generations of development:

The First Generation (1938-1953): This generation was marked by the use of very slow electronic components (switching devices) such as relays in the 1940's and vacuum tubes in the 1950's. The first electronic digital computer ENIAC (Electronic Numerical Integrator and Computer) was introduced in 1946. In 1950 the first-stored-program computer, EDVAC (Electronic Discrete Variable Automatic Computer), was developed.

The Second Generation (1952-1963): The introduction of the first transistorized digital computer, TRADIC, built by Bell laboratories in 1954, marked the beginning of the second generation of computers. Discrete transistors and diodes were the building blocks. In 1960, IBM introduced its first scientific, transistorized computer,
IBM1620. COBOL (Common Business Oriented Language) was developed in 1959. Magnetic core memory was developed and used in many machines.

The Third Generation (1962-1975): This was marked by the use of small-scale integrated (SSI) and medium-scale integrated (MSI) circuits as the basic building blocks. Multilayered printed circuits were used. By 1968, many fast computers began to replace magnetic cores with semiconductor memories. Many high-performance computers, like IBM 360/91, Illiac IV, cyber-175, and c.mmp, and several vector processors were developed in the early 70's. High-level programming languages (e.g. COBOL, FORTRAN) were enhanced with intelligent compilers, and sophisticated operating systems with time-sharing feature became available during this period.

The Fourth Generation (1972-present): In this generation, large-scale integrated (LSI) circuits are used for both logic and memory sections, with high-density packaging in construction. Computers are characterised by enhanced levels of circuits integration through VLSI techniques. Examples of available high-speed mainframes are Cray X-MP, IBM 370/168, IBM 3081, and Fujitsu M382. Various software and architectural innovations have been introduced to support a wide range of diverse applications.

Thus, any further increase in computation speed can only be obtained through both increased switching speeds and increased circuit densities. The current technology has gone a long way in this direction and the development continues. Nowadays complex digital electronic systems can be fabricated on a single chip of silicon, one-tenth the size of a postage stamp. In fact, it is foreseen that the
number of components that a VLSI chip could accommodate would be increased by a multiplication factor of ten to one hundred in the next two decades [93]. However, there is of course a natural limitation in technology development, and any breakthroughs in the future will be difficult and costly. The size of wires and transistors approach the limits of photolithographic resolution and it will be literally impossible to achieve further miniaturization. Even if switching times become instantaneous, distance between components may not become small enough to decrease generation delays enough to make a real increase in computation speed [123].

To handle this problem, new approaches must be considered. One way of increasing the speed of computation is to perform as many operations as possible simultaneously, in parallel, instead of sequentially. It is hoped that, by connecting together \( m \) identical processing elements, the speed up achieved is approximately \( m \) times that of a single processor.

2.2 Classification of Designs

In this section we will present the concepts of the architectural classification given by different researchers, especially by Flynn [40], Kuck [83], Treleaven [128], and Gajski and Peir [46].

2.2.1 Flynn's TAXONOMY

Flynn's classification of computer systems is based on how the machine relates its instructions to the data being processed, rather than on the structure of the machines [40]. According to Flynn, digital computers may be classified into four categories according to the multiplicity of instruction and data streams, (see Figure
2.1). The term stream, as it is used here, means a sequence of items (instruction or data) as executed or operated on by a single processor. Listed below are Flynn's four machine organization:

- Single instruction stream / single data stream (SISD)
- Single instruction stream / multiple data stream (SIMD)
- Multiple instruction stream / single data stream (MISD)
- Multiple instruction stream / multiple data stream (MIMD)

(i) **SISD computers**: This is the well known sequential computer, or uniprocessor system (see Figure 2.1a). Examples of SISD computers are: IBM701 (Von Neumann architecture computer); CDC 6600, IBM 360/91 (computers with multi functional units), and VAX 8600, CDC 7600 (pipelined computers).

(ii) **SIMD computers**: This class corresponds to both vector or array processors. As illustrated in Figure(2.1b) each instruction operates on a data vector or array rather than on a single operand. The multiple processing elements are supervised by the same control unit. Examples of SIMD systems are ILLIACIV, ICL DAP, and MPP.

(iii) **MISD computers**: This class is illustrated in Figure 2.1c. In this system, there are several instructions operating on the same data stream. The conventional view is that such a machine has not yet appeared, although there is also a view that pipelined vector processors belong to this category, rather than SIMD as they are classified.

(iv) **MIMD computers**: In the MIMD system we have a set of \( n \) processors connected together and simultaneously executing different instructions on different sets of data (Figure 2.1d). Each of these processors is a complete computer. This class
includes all the different multiprocessors configurations, starting from the linked main-frame systems, up to large arrays of microprocessors. Examples of this type are the Sequent Balance 8000/21000, the Transputer, the Cosmic Cube, the Ultracomputer and the BBN Butterfly. There are problems with Flynn's classification, since it groups all parallel computers except the multiprocessor into
the SIMD class, and considers, pipelined and array processor computers as similar, although they are two completely different architectures.

2.2.2 OTHER CLASSIFICATIONS

In this section we present some other classification approaches. One of these was suggested by Kuck [83]. Kuck replaces the data streams, in Flynn’s classes, with execution streams, and then considers scalar and array instruction streams as well as scalar and array execution streams. He identified 16 categories which are shown in Figures (2.2 a) and (2.2 b). For example, machine type (2) is SIS SEA, which means Single Scalar Instruction stream, Single Array Execution stream. Kuck’s taxonomy gives more description at the hardware level, whereas Flynn’s is more at the level of the architecture or the instruction set. Machines like Cray X-MP and Ultracomputer would belong to Flynn’s MIMD class, whereas for Kuck these would be separated into two different categories, namely the MISMES and the MISMEA. The Ultracomputer processing elements have scalar instruction sets whereas the Cray includes array instructions. Although Kuck’s taxonomy is more detailed than that of Flynn, the second is more widely used.

In addition, Treleaven [128] presented a further classification of the MIMD class. The data mechanism was divided into shared-memory (Von Neumann-like) and private-memory (message-passing) approaches. For the control mechanism, he listed four explicit forms: Control driven, Pattern driven, Demand driven and Data driven. Consequently the MIMD class was subdivided into 8 subclasses (see Figure 2.3):
Figure 2.2a: Kuck's classification of parallel architectures.

1. Control driven machine with shared memory.
2. Control driven machine with private memory.
3. Pattern driven machine with shared memory.
4. Pattern driven machine with private memory.
5. Demand driven machine with shared memory.
6. Demand driven machine with private memory.
7. Data driven machine with shared memory.
8. Data driven machine with private memory.

An operation can be triggered by the arrival of an instruction, a demand for an operation’s result, or arrival of the data needed by an operation. The control driven is often referred to as the Von Neumann approach to parallel processing. It consists of interconnecting two or more Von Neumann-type uniprocessor each of which,
synchronously or asynchronously, follow the traditional program sequence -
controlled cycle of fetch-execute-store using global and/or local memories. The data-

14
driven concept is the data flow approach. Its basic idea is to enable the execution of an instruction whenever its required operands become available (non-sequential operation). Thus, no program counters are needed in data-driven computations.

Finally the demand driven approach is called the reduction approach. It consists of carrying out instructions when results are needed for other calculations. Programs are viewed as nested applications (calculations), and execution proceeds by successively reducing innermost applications according to the semantics of their respective operators until there are no further applications. Calculations are performed only when they are needed, and not whenever their operands are ready, as in dataflow.

![Diagram showing MIMD architectures]

Figure 2.3: Treleaven's classification of MIMD architectures by control and data mechanism.

Gajski and Peir [46] take Treleaven's shared memory MIMD category and subdivided it further according to the granularity of the parallelism. They consider parallelism at three levels: the tasks, the processes, and the instructions levels. They define tasks as units of scheduling that can be assigned to one or more
processing elements (PEs). A process is part of a task performed on only one PE. Thus, the overall job of executing a program can be viewed as a collection of tasks. Three different architectures are identified according to the control mechanism used at several levels of program granularity.

1. The first class is characterised by a parallel control mechanism at the process and instruction levels, and a serial control at the task level. (Cray X-MP).

2. For the second class, a parallel control mechanism is used at the task level, and a serial mechanism at the instruction level. (Ultracomputer).

3. Finally, the third class is characterised by a parallel control mechanism at the task and process levels, and serial mechanism at the instruction level (Cedar).

There are many other classifications available, but the problem of finding one classification that includes all the computer architectures remains unresolved. For information about other classification approaches we recommend Feng [38], Händler [62], Shore [115], Basu [21, 22], Johnson [76] and Skillcorn [117].

2.3 Structural Classification of Computer Systems

Hockney and Jesshope [71], summarized the principal ways of introducing parallelism into the architecture of computers as:

(i) The application of pipelining techniques to improve the performance of arithmetic or control units.

(ii) Providing several functions, such as logic, addition and multiplication, where these functions operate in parallel on different data.
(iii) Providing an array or vector of processing elements performing the instruction simultaneously, but on different data, where the data is stored in the processing elements private memories.

(iv) Providing several processors each of them being a complete computer. Also they show by historical discussion that parallelism has been applied at several distinct levels which might be classified as:

**Job level**
(a) between jobs (multiprocessing and multiuser).
(b) between phases of one job (multitasking).

**Program level**
(a) between parts of a program (multitasking).
(b) within the DO loops (large grain parallelism).

**Instruction level**
(a) between phases of instruction execution. (medium grain parallelism).

**Arithmetic and bit level**
(a) between elements of vector arithmetic. (small grain parallelism).
(b) within arithmetic logic circuits. (hardware).

With the aid of the above discussion and the structural notation of computers, Hockney and Jesshope [71] formulated a structural classification of computer systems (serial and parallel computers). The main subdivisions are shown in Figures (2.4) and (2.5) together with a well-known example in each class. Their taxonomy was more detailed than that of Flynn and took implicit account of pipelined structures. Therefore the multiple instruction class was not considered for further categorisation as with the pipelined and array processor computers.
Figure 2.4: Structural classification of computers

Figure 2.5: Parallel Unicomputers based on Functional Parallelism and Pipelining
Nevertheless, this scheme if coupled with that of Flynn could well be suited for a general classification of parallel computers.

2.4 Parallel Computer Structures

Parallel computers are those systems that emphasize parallel processing. We are going to present three architectural configurations of parallel computers, these are:

- Pipeline computers
- Array processors
- Multiprocessor systems

A pipelined computer performs overlapped computation to exploit temporal parallelism.

An array processor uses multiple arithmetic logic units in a synchronized manner.

A multiprocessor system achieves asynchronous parallelism through a set of interactive processors with shared resources (memories, database...).

In addition to these three concepts, there are new approaches to parallel processing such as Data flow, Reduction and VLSI approaches. Data flow computers were suggested in recent years. Their basic concept is to enable the execution of an instruction whenever its required operands become available (non-sequential operation). Thus, no program counters are needed in data-driven computations. The reduction approach consists of carrying out instructions when results are needed for other calculations. Programs are viewed as nested calculations, and execution proceeds by successively reducing innermost calculations. So calculations
are performed only when they are needed and not whenever their operands are ready, as in dataflow. The main idea of VLSI is to implement enormous numbers of devices on a chip. The rapid advent of very-large-scale integrated (VLSI) technology, has created a new architectural horizon in implementing parallel algorithms directly in hardware. In addition, it will be possible to implement SIMD and MIMD architectural designs on a chip.

In this chapter attention will be concentrated on multiprocessor systems, while giving a brief idea about pipeline computers and array processors.

2.4.1 PIPELINE COMPUTERS

Pipelining is essentially operational management technique, which is based on subdividing the job into tasks, each to be assigned to separate working units. Händler [62] classified the pipeline processors into three classes according to the levels of processing. The three classes are: Arithmetic pipelining, Instruction pipelining and Processor pipelining. A simplified model of the pipeline machine is shown in Figure 2.6. This model consists of \( p \) pipelining segments. Each segment executes a part of the processing and the result appears at the end of the last segment.

As an example, consider the process of executing an instruction. Normally, the process involves four major steps: instruction fetch (IF), instruction decoding (ID), operand fetch (OF) and finally the execution (EX).

In a non-pipelined computer, these four steps must be completed before the next instruction can be issued. In a pipelined computer, successive instructions are
executed in an overlapped fashion. The four pipelined stages are connected serially as shown in Figure 2.7a. After constant time intervals, the output of one stage is shifted to the next. For a nonpipelined (non-overlapped) computer, it takes four pipeline cycles to complete one instruction. In a pipelined computer
once a pipeline is full up, an output result is produced on each cycle. So by overlapping the execution, the instruction processing will be faster by a factor of four over non-pipelined execution. The two space-time diagrams (Figure (2.7b,c)) show the difference between overlapped instruction execution and sequentially non-overlapped execution.

![Diagram of a pipelined processor](a) Pipelined processor

![Space-time diagram for a non-pipelined processor](b) Space-time diagram for a non-pipelined processor

![Space-time diagram for a pipelined processor](c) Space-time diagram for a pipelined processor

Figure 2.7 Basic concept of pipelined processor, overlapped and non-overlapped instruction execution

22
Pipeline computers are more attractive for vector processing, where component operations may be repeated many times. Most existing pipeline computers emphasize vector processing.

A typical pipeline computer is shown in Figure 2.8. This model shows both scalar arithmetic pipelines and vector arithmetic pipelines. The instruction processing unit is itself pipelined with three stages as shown.

Examples of pipelined computers are: CDC STAR-100, CRAY-1 and Texas Instrument's Advanced Scientific Computer (ASC). Detailed descriptions of pipeline computers can be found in Hwang [72].

2.4.2 ARRAY PROCESSORS

Array processors can be defined as a synchronous array of parallel processing elements (PEs), that can operate in parallel in a lock-step fashion. The PE's are controlled by a single control unit. There are two essential reasons for building array processors. The first is economic, in that $N$ processors with only a single control unit, are cheaper than $N$ computers. The second reason concerns interprocessor communication, the bandwidth can be more fully utilised. A typical array processor is depicted in Figure 2.9 scalar and control-type instructions are directly executed in the control unit (CU). Each PE consists of an Arithmetic Logic Unit (ALU). The PE's are interconnected by a data-routing network. The PE's are synchronized to perform the same function at the same time. The control unit decodes the instruction and broadcasts the instruction via control lines to all PE's simultaneously. Thus, a common instruction is executed by all PE's simultaneously using data from its local memory. Instruction fetch (from local memories or from
the control memory) and decode is done by the control unit. The PE's are passive devices without instruction decoding capabilities. Various interconnection networks have been suggested for array processor computers. Obviously, a complete network,
where each processor is connected to all other processors is expensive and unmanageable by both the designer and the user of the system. Therefore, some other interconnection patterns are proposed to be of primary importance in determining the power of parallel computers.
The interconnection networks can be generally distinguished into two types: the bus and the alignment networks. The former allows only a single one-to-one communication to take place at any given time. The latter allows several one-to-one or one-to-many communications. The bus network is less expensive than the alignment network but slower than it.

Furthermore, the alignment networks can be classified into static and dynamic networks [72]. A static network is characterised by the required dimensions for layout. For example a one, two or three-dimensional and hypercube as shown in Figure 2.10.

On the other hand, the dynamic networks are distinguished into single-stage networks and multi-stage networks as shown in Figure 2.11.

To run a program efficiently, it is necessary to use a suitable interconnection pattern for the problem algorithm and to use as many processors as possible. In the ICL DAP or ILLIAC IV, the arithmetic processors are arranged in a two-dimensional array where each processor is connected to its nearest 4 neighbours as illustrated in Figure 2.12. This type of communication network is suitable to solve two-dimensional partial differential equations. Stone [122], proposed an interconnection pattern called the perfect shuffle (see Figure 2.13), where applications such as Fourier Transforms, sorting and matrix transposition make effective use of it. Goldschlager [54], proposed a universal pattern which can simulate all these patterns.

Before finishing this section we shall talk about associative memory and associative processors. Data stored in an associative memory are addressed by
their contents. In this sense, associative memories have been known as content-addressable memory, parallel search memory, and multi-access memory. The major advantage of associative memory over conventional random-access memory (RAM)
is its capability of parallel search and parallel comparisons. So the fundamental distinction between associative memory (AM) and RAM is that the AM is content-
addressable, allowing parallel access of multiple memory words, whereas the RAM must be sequentially accessed by specifying the words addresses. Array processors designed with associative memories are called associative processors. An associative
processor has the following properties: (1.) stored data items are content-addressable, and (2.) arithmetic and logic operations are performed over many sets of arguments in a single instruction.

Array processors are effective in many special application areas mostly non-numerical such as radar signal tracking and processing, weather prediction computations and many types of information processing.

We can classify associative processors into four categories based on the comparison process followed by the associative memory: the fully-parallel, the bit-serial, the word-serial and the block-oriented associative processors. The Parallel Element Processing Ensemble (PEPE) and STARAM are the best-known fully parallel and bit-serial associative processors respectively.

2.4.3 MULTIPROCESSOR SYSTEMS

Multiprocessor systems were developed as a result of the efforts and research work aiming to improve system speed, reliability, throughput, flexibility and availability. Multiprocessor systems are an alternative design of great promise to produce a high speed or super computer. These are computers that reach the performance goals that scientists would like to achieve and use for the next generation of machines. VLSI technology made replication of hardware units affordable.

Multiprocessor architectures use a number of identical processors in parallel to obtain high execution speed. The multiprocessor approach introduces three new requirements which have not been encountered in the uniprocessor environment. First, each problem should be partitioned into tasks. Second, each task should
be scheduled for execution on one or more processors. Third, synchronization of control and data flow must be performed during execution. In addition, program partitioning introduces several levels of execution control, and parallelism can be exploited on each leading to different hierarchical control structures.

Multiprocessor system organization is determined primarily by the interconnection patterns between processors, memory modules and peripheral devices. The processors need to communicate so that they can cooperatively solve a given problem. This communication can be achieved through a shared memory or by using a message-passing mechanism. In the first case, the system is called tightly-coupled multiprocessor, and in the second it is called loosely-coupled multiprocessor.

2.4.3.1 Interconnection Networks

The techniques selected to connect processors to each other, to memory modules and to peripheral devices, in a multiprocessor system, is one of its most important architectural features and has profound impact on system capabilities, performance, size and cost. Most commercial machines have used time-shared common buses, crossbar switch networks, multiported memories, or the hypercube topology for interconnecting processors and memories. The most widely used method of connecting processing elements, memory units and peripherals to each other is the bus structure. Network topology plays an important role in the way data structure are distributed around the system. On networks with a low bisection width, such as a tree, certain data movements are very slow. A complete study of the role of topology in parallel algorithm design is found in Gannon and Van Rosendale [48].
In the following subsections we present the bus structure, the crossbar networks, and the multiport memories. Information about other interconnection networks such as single-stage and multi-stage networks, and the hypercube topology is found in Decegama [32].

(i) *Time shared common bus:* The time shared common bus interconnection scheme, represents the simplest form of connecting all the functional units using a single bus. It consists of a common communication path connecting all the functional units. The transfer operations are controlled by the bus interfaces of the sending and receiving units. Since the bus is a shared resource, mechanism must be provided to resolve contention. The processor or the I/O unit wishing to initiate a transfer, must first determine the availability states of the bus, and then address the receiving unit as well as to determine its availability and capability to receive the transfer.

By its nature, the single-bus organization is quite reliable and relatively inexpensive. However a single malfunction in any of the bus interface units can cause complete system failure and the total overall transfer rate. Moreover, system expansion, by adding more processors or memory, increases the bus contention, which degrades system performance and increases arbitration logic. The total overall transfer rate within the system is limited by the bus bandwidth and speed. Several interconnection systems such as the use of two one-way paths and multiple two-way buses have been provided for throughput or reliability reasons. (see Figure 2.14).
(ii) **Crossbar switch network:** The crossbar switch is used to overcome the inadequacies of the time-shared bus organization. In the crossbar switch system, the number of buses is increased to reach a point where there is a separate path
available for each memory unit as shown in Figure 2.15. The main characteristics of the crossbar switch are high throughput, easy to isolate the malfunctioning device, the addition of functional units to attain improved system performance and most expensive and extensive interconnection system. The crossbar switch has the potential for the highest bandwidth and system efficiency. However, because of its complexity and cost, it may not be cost-effective for a large multiprocessor system.

![Crossbar switch system](image)

Figure 2.15: Crossbar switch system
An example of a crossbar interconnection system is the Carnegie-Mellon miniprocessor system (C.mmp), which has 16 processors and 16 memory modules.

(iii) **Multiport memories**: Multiported memory systems are the result of concentrating the control, switching and priority arbitration logic (which are distributed throughout the crossbar switch matrix), at the interface to the memory units (see Figure 2.16). Memory-access conflicts are resolved by assigning permanently designated priorities at each memory port (see Figure 2.17). It is possible also to designate portions of memory as private to certain processors, I/O units, or combinations of both. Figure 2.18 shows a multiport memory organization with private memories. In this Figure, memory modules $M_0$ and $M_3$ are private to processor $p_0$ and $p_1$, respectively.

This organization offers a high potential transfer rate within the system, but with severe constraints on the number of processors imposed by the number and type of the memory ports.

Univac 1100/90 and IBM system 370/168 are both examples of multiport memory systems.

Finally, we discuss one of the factors that degrade the overall performance of the system, namely the interference or conflict, produced in the accessing of a shared memory. The memory access conflict can be generally classified into two types: software and hardware types.

The software memory conflict occurs when a processor attempts to use data that is currently being accessed by another processor which has activated a software
‘lock’ mechanism to prevent any other processor from accessing the same data set. This data set is called the "critical section".

The hardware type of memory conflict is caused when two or more processors

![Multiport memory organization](image1)

Figure 2.16: Multiport memory organization

![Multiport memory system with assignment of port priorities](image2)

Figure 2.17: Multiport memory system with assignment of port priorities
Figure 2.18: Multiport memory organization with private memories

attempt to access the same memory module simultaneously, i.e. the conflicting requests are made to the same module during a single memory cycle by different processors. Therefore, only one access can be made per memory cycle and the other requests must wait usually for one cycle or two in each case. Thus, programs with a large number of these conflicts have greater degradation in their overall performance. To reduce the memory conflicts and the processor interconnection, an increased use of private memory which is associated with each processor is then suggested to store important data.
2.4.3.2 Functional Structures

Multiprocessors may be classified into two main classes: the tightly-coupled and the loosely-coupled systems depending on the degree of communication or coupling that exists between the processing elements (see Figure 2.19).

![Diagram of Multiprocessor Systems]

(a) Tightly-Coupled system

(b) Loosely-Coupled system

Figure 2.19: Multiprocessor System
(i) *Loosely-coupled multiprocessors*

In loosely-coupled systems (LCS) each processor has a set of input-output devices and a large local memory where it accesses most of the instructions and data. We refer to the processor, its local memory and I/O interfaces as a "computer module". A loosely-coupled multiprocessor is formed by a number of such computer modules which communicate by the means of "Message Transfer Systems" (MTS), at I/O level. Therefore, the integration in LCS is achieved by communication between the I/O systems. In this way programs and data can be shared among processors. In LCS, main memory is not shared among processors. Processors which execute on different computer modules communicate by exchanging messages through the MTS. The communication topology of the associated message-transfer system is the determinant factor of the degree of coupling in LCS. Usually LCS's are efficient when the interaction between tasks are minimal.

Other characteristics of LCS are:

1. In LCS, one of the processors is designated as overall system control (global processor). The other processors are called local processors. All jobs enter the system through this global processor and in the event that the global processor fails, one of the local processors may act as global processor.

2. In LCS, problems such as synchronization, communication, data transfers, task partitioning, and software control has to be taken under consideration. The user specifies the way to divide the tasks among the modules in order that they can operate concurrently.

3. Each module can stand by itself.
An example of a nonhierarchical loosely-coupled multiprocessor system is shown in Figure 2.20. Each computer module consists of a processor, a local memory, local I/O devices and an interface to other computer modules which may contain a channel and arbiter switch. The arbiter switch role is to choose, in the case of two or more modules request to access a physical segment of the MTS, one of the simultaneous requests according to a given service strategy. It delays other requests until the selected request has been served. The channel may have a high-speed communication memory used as a buffer to store block transfer of messages. The communication memory is accessible to all the processors.

![Diagram of a nonhierarchical loosely coupled multiprocessor system](image)

Figure 2.20: Nonhierarchical loosely coupled multiprocessor system

A very important factor that determines the performance of the multiprocessor system is the message-transfer system (MTS). It could be a single time shared bus, or a shared memory system. For LCS with a shared memory MTS is the limiting factor in the memory conflict problem imposed by the processor-memory interconnection network.

In the case of LCS using a single time shared bus, the performance is limited by the message arrival rate on the bus, the message length and the bus capacity.
The communication memory may also be centralized and connected to a time shared bus, or be part of the shared memory system. There are two sorts of processes (tasks) communication, the first is between processes allocated to the same processor, and the second between tasks allocated to different processors. An input port, stored in the local memory of the processor to which the task is allocated, is associated with each process. If a task wants to issue a message to another task, it places the message into the input port of the destination task. Communication between tasks allocated to the same processor takes place through the local memory only, while communication between tasks allocated to different processors is through a communication port residing in the communication memory. One communication port is associated with each processor as its input port. With the advent of VLSI technology, the computer module can be fabricated on a single integrated circuit and be used as the building block of a multiprocessor system.

Finally, loosely-coupled systems can be characterised by: the speed of the floating-point unit, the start-up time for I/O operation, and the transfer rate. These three parameters are measured in seconds per operations. Examples of loosely-coupled systems include PASM, the hypercube and the transputer.

(ii) Tightly-coupled multiprocessors

Tightly-coupled systems can be defined as a large number of processors sharing a common parallel memory (the main memory), and controlled by a single operating system. The processors communicate with each other through the shared main memory. Hence the rate at which data can communicate from one processor to the other is of the order of the bandwidth of the memory. A small local memory
or high-speed buffer (cache) may exist in each processor. Cache memories (high-speed buffers) are inserted between the processor and main memory to capture those portions of the contents of main memory which are currently in use. Since cache memories are typically five to ten times faster than main memory, they can reduce the effective memory access time if carefully designed and implemented. A complete connectivity exists between the processors and the shared memory. This connectivity can be accomplished either by inserting an interconnection network between the processors and the memory or by a multiported memory. Contrary to Loosely-coupled systems which are usually efficient when the interactions between tasks are minimal, Tightly-coupled systems (TCS) can tolerate a higher degree of interactions between tasks without significant deterioration in performance. In loosely-coupled systems the throughput may be too low for some applications that require fast response times. If high-speed or real-time processing is desired tightly-coupled systems may be used.

For tightly-coupled multiprocessors, the operating system resembles that of a uniprocessor with multiprogramming capability. However, it is more complex, since the processes (tasks) can be assigned to more than one processor.

In order that the processors can access the memory simultaneously, it is desirable that it is partitioned into independently addressable modules. Figure 2.21 shows two typical models of a TCS, the first model (Figure 2.21a) is without cache memories, and the second (Figure 2.21b) is with the use of cache memories.
Both models consist of $p$ processors, and $d$ input-output channels. In the first model, the shared memory is partitioned into $\ell$ modules, while the second model has a pipelined shared memory modules. These units are connected through a set of three interconnection networks namely, the processor-memory interconnection network (PMIN), the I/O-processor interconnection network, and the interrupt-signal interconnection network. The processor-memory interconnection network is a switch which can connect every processor to every memory module.

To avoid excessive conflicts, the number of memory modules $\ell$, in the first model, is usually as large as $p$. Another method is to associate an unmapped local
memory (ULM) with each processor. This ULM is a reserved storage area, used to store kernel code and operating system tables often used by the processes running on that processor. The memory map (MM) is a module, attached to each processor and which directs the memory references to either the unmapped local memory (ULM) or the private cache of that processor.
In Figure 2.21b a cache is associated with each processor. Each processor may make memory references which are accessed in main memory. These memory references contribute to the memory conflicts at the memory modules. To reduce this problem (delay) the cache is associating with each processor to capture most of the references made by a processor. On the other hand, using the cache can reduce the traffic through the crossbar switch (in PMIN), which subsequently reduces the contention at the cross points.

Finally, a limiting factor to the expansion of a tightly-coupled system is the performance degradation due to the memory contentions which occur when two or more processors attempt to access the same memory unit concurrently. Another limiting factor is the processor-memory interconnection network itself.

Examples of tightly-coupled multiprocessors include the Sequent Balance system, the Multimax system, the BBN Butterfly and the NYU Ultracomputer. The Balance system developed and commercialised by Sequent Inc., is described in the next section.

2.5 The Sequent Balance Multiprocessor System

2.5.1 SYSTEM HARDWARE

The Balance is a shared-memory, tightly coupled multiprocessor system. This system can contain two to thirty 32-bit microprocessors (two to twelve in the Balance 8000, two to thirty in the Balance 21000), with an aggregate performance of up to 21 million instructions per second (MIPS). Later versions of the system include the symmetry systems 27000 and 81000. The processors share a common memory
pool and a single copy of Dynix, a multiprocessor operating system supporting both
the 4.2BSD and system V Unix environments. Each processor has a private cache
as well as a small local memory to hold frequently used kernel routines. The system
contains a high-bandwidth pipelined bus a diagnostic and console processor, up to
28 Mbytes of main memory, up to four IEEE 769 (Multibus) adaptors, an IEEE
802.3 (Ethernet) LAN interface, and an ANSI Small Computer Systems Interface
(SCSI).

In this section we shall describe the Sequent Balance 8000.

The Balance 8000 is a general-purpose parallel computer which support simulta-
taneous execution of parallel programs and existing sequential applications.

From the hardware point of view, the Balance 8000 is a member of a family of
products that implement a scalable processor pool architecture. It consists of a pool
of two or twelve tightly coupled processors, a bandwidth bus, up to 28 Mbytes of
main memory, up to four high-performance I/O channels, a diagnostic and console
processor, and up to four IEEE 769 (Multibus) bus couplers (see Figure 2.22).

Each processor is itself a subsystem, packaged two per circuit board (see Figure
2.23). A subsystem includes three VLSI components: a 32-bit N.S. 32032 CPU, a
32081 hardware floating-point unit and a 32082 paged virtual memory management
unit.
Each processor subsystem includes a cache memory that almost reduces to zero the waiting periods of all the processor while minimizing bus traffic. Cache memories are six times faster than main memory; they reduce the effective memory access time and match the high-speed CPU with the lower speed memory. The Balance two-way, set-associative cache provides 8 kbytes of very high-speed buffer memory and stores recently accessed instructions and data, so that subsequent requests for this information are satisfied from the cache rather than main memory.

The association of a private cache with each processor introduces the problem of cache consistency. This is the problem of keeping data coherent among the caches. To solve the cache coherence problem, Balance system includes a set of bus-watching logic implemented in every cache.
This set of bus-watching logic continuously monitors write cycles on the system bus and compares the write addresses with those in the cache so that it can detect hits to cache entries. When a hit occurs in the bus-watching logic, the cache controller
invalidates the affected entry. Another coherence problem is that of the coherence of the data between the main memory and the caches on each processor. To ensure that the main memory datum is updated whenever a cached copy is updated, the Balance system utilises a "write-through" cache consistency mechanism. In addition to updating the cache, each write cycle goes through to the bus and memory. A writer buffer is implemented in each processor to avoid performance impact resulting from the time spent by the processor waiting for the completion of each bus write cycle. The processor issues a write cycle and proceed immediately while the write buffer independently completes the main memory cycle. The last component of the processor subsystem is the System Link and Interrupt Controller (SLIC). SLIC is a chip and it is associated with each major component in the system i.e. processors, memory controllers, I/O controllers, etc. This SLIC chip manages interprocessor communication, synchronised access to shared data structures, distribution of interrupts among the processors, and diagnostics and configuration control. An SLIC provides two basic kinds of interrupts, maskable and nonmaskable, which correspond to the interrupt request lines present on most microprocessors. Interrupts are further characterised by the way they are directed. Two kinds of interrupts can be distinguished: interrupt directed to a particular SLIC (directed interrupt), and interrupt directed to one of a group of SLICs (group interrupt). In the latter, the selection of which group is an initialisation parameter to the SLIC. Each SLIC implements a set of 64 binary semaphores, called gates, and supports a set of SLIC commands to test and set them. Each SLIC contains a copy of the value of each of the 64 gates at all times.
The SLIC bus which is part of the Balance 8000 system bus (SB8000) provides an interconnection for communication among the SLIC chips.

The SB8000 system bus is a critical element of the system. It provides software-transparency, i.e. symmetric access from all processors to all system resources, including main memory and I/O subsystems of widely varying access latency. The system bus is a 32-bit wide, pipelined, packet bus supporting multiple overlapped memory and I/O transactions. It supports several packet lengths and checks parity to aid in error detection.

All the processors share the same memory. The system supports up to 28 Mbytes of principle memory, a 4 Mbyte I/O address space, and a 16 Mbyte virtual memory address space for each process. The Balance 8000 supports up to four memory controllers, each with an optional expansion board, reducing contention for memory among the processors.

The Balance supports standard I/O throughout the system, and also permits several instances of each interface to increase the I/O bandwidth.

The Balance 8000 also supports:

(i) Up to 4 SCSI/Ethernet interfaces for disk and tape I/O and communication with other systems.

(ii) Up to 4 MULTIBUS interfaces for serial communications and large disk and tape support, and user-added devices.

2.5.2 SYSTEM SOFTWARE

From the systems software, Dynix, Sequent’s multiprocessor implementation of Unix, is Balance Operating system. All the processors share a single copy of
Dynix. Dynix is an enhanced Unix Operating system. The enhancements have been made in five major areas of the operating system model: the mutual exclusion, the interrupt distribution, the process scheduling, the shared-memory management and the virtual memory management.

The Dynix kernel or executive has been made shareable so that multiple processors can execute the same system calls and other kernel code simultaneously. The Dynix system controls the assignment of processes to processors, dynamically balancing the running tasks. Any process can run on any processor at anytime (provided that processor has not been made dedicated to another task).

Dynix determines the minimum and maximum amount of physical memory a process can consume. It then adjusts the memory allocation for each process between these bounds to maintain each process's paging rate and tune the virtual memory performance for the entire system.

The multiprogramming features of UNIX such as pipes and forks are retained, and in addition the Dynix and the parallel programming library supply the fundamental parallel programming mechanisms: process creation and termination, interprocess communication and synchronisation via shared memory, and Unix signals and mutual exclusion via spinlocks.

The current Dynix operating system includes both 4.2 BSD and system V command libraries.

Parallel programs are difficult to debug when using conventional debuggers that monitor only one process at a time. The sequent pdbx parallel debugger allows the programmer to execute a parallel program in a controlled environment in which all the streams of executing can be monitored.
Finally, as measure of system performance, the multiuser benchmarks show that a fully configured Balance 8000 with up to 12 processors can support up to 96 users, and a Balance 21000 system with up to 30 processors can support up to 256 users.

2.6 Parallel Computer Architectures for Artificial Intelligence (AI)

2.6.1 CHARACTERISTICS OF AI COMPUTATIONS

To develop a special-purpose computer for AI applications, we should first understand and be able to define the main requirement for this application. In contrast to numerical applications, AI applications are characterised by symbolic processing, nondeterministic computations, dynamic execution, large potential for parallel and distributed processing and finally the management of huge knowledge bases. AI processing requires large memory with intensive access patterns. Von Neumann architecture with centralised control presents a processor/memory bottleneck to intensive and irregular memory access patterns, and consequently this machine fails to provide the execution speed demanded by AI applications. An AI computer system should support both knowledge-based management and heuristic searches, and make use of faster technologies and parallel processing techniques to improve the computational efficiency. Some AI tasks require intensive computation, and it is becoming more necessary to use parallelism in order to process these tasks efficiently. Many parallel architectures have been designed for a variety of AI applications. Examples of parallel AI systems currently available include: ALICE, Butterfly, Concurrent Lisp machine, Connection machine, DADO, FAIM-1, NONYON, SOAR, and ZMOB (see Wah [134]). Recently, small grain massively parallel
architectures, called the connectionist systems are being developed. These systems include a large number of simple processing elements, and use connections rather than memory cells to store information [37].

An AI machine can be designed with a top-down, middle-out, or bottom-up methodology. In the top-down method, designers begin with specifying the requirement for the application being implemented and gradually works down toward the hardware level. The middle-out methodology starts with a well-established language or knowledge representation scheme and then works in both upward and downward directions at the same time. In the upward direction, the work involves the modification of the language to make it suitable for the application. The work in the downward direction involves mainly the design of the hardware architecture to support the language used. Finally in the bottom-up method, designer starts with the selection of a computational model (such as control flow, data flow, or reduction) and then move to designing an architecture to implement it. The last step is to build the software, on top of the hardware, to support the application.

Finally, the high performance of many parallel AI computers is achieved through associative processing and data-level parallelism. This approach is used for applications involving operations on large databases, such as sorting, set operations, statistical analysis, and associative pattern matching. For other AI applications involving heuristic searches, control-level parallelism is used.

In the remainder of this chapter we are going to describe four major classes of AI machines namely: language-based, knowledge based, connectionist and intelligent interface machines. This is followed by a description of the DADO production system machine.
2.6.2 LANGUAGE-BASED AI MACHINES

Language-based AI machines are mainly designed to efficiently execute LISP, PROLOG, or a functional language. The knowledge representation and inference mechanisms should be written in the same language supported by the hardware. So, for a LISP machine, the knowledge representation and the inference mechanism must be written in LISP. Most present commercial Lisp machines use the middle out design approach. Symbolics 3600 series [99] is an example of these machines. Another important implementation based on very large scale integration (VLSI) architecture, is called Lisp on a chip implementations. The major impacts of these chips is to increase the availability of Lisp machine by lowering the cost while increasing their performance. The Japanese Fifth Generation Computer System Project's PIM [126] and PIE [100] architectures, are the only top-down implementations of language-based machines. Common Lisp chips [69], Tamura machine [125] and ZAPP [132] are examples of language-based AI machines that use the bottom-up implementation approach. The ALICE machine implements the functional language HOPE using graph reduction on a shared-memory multiprocessor [30]. The Xenologic X-1 [34] is a coprocessor board for the Sun workstation that implements Prolog with extensions for Lisp. The Tamura machine, on the other hand, implements Prolog on a conventional multiprocessor.

2.6.3 KNOWLEDGE-BASED AI MACHINES

Knowledge-based AI machines are developed to support a particular knowledge representation such as: semantic networks, rules, frames, or objects. (See Chapter
4 for details about these knowledge representations). Consequently, three types of
knowledge-based machines can be identified, these are:

1. Rule-based machines.
3. Object-based machines.

DADO2 [120], NON-VON [113] and PSM [59] are examples of rule-based ma­
chines. Both DADO2 and NON-VON are massively parallel machines which are
intended to be implemented with tens of thousands of 8-bit processors. The PSM
uses 32 to 64 very powerful ECL-based processors.

The Connection Machine [67] and SNAP [98] are examples of semantic networks
machines. The Connection Machine is a massively parallel fine-grain machine, while
SNAP is an array-structured machine for processing semantic networks and is de­
signed for VLSI implementation. Finally, as examples of object-based machines we
consider: FAIM-1 [9] and SOAR [131]. The FAIM-1 is a multicomputer system
with a unique hexagonal mesh topology chosen for its fault tolerance, scalability
and planar characteristics. This machine implements both logic and procedural
programming within an object-oriented frame work. On the other hand, SOAR
implements Smalltalk-80, the archetypical object oriented language.

2.6.4 CONNECTIONIST MACHINES

Connectionist machines are the only AI machines which do not base their ar­
chitectures on symbolic processing. On such machines, knowledge is not anymore
represented by symbols, but instead it is encoded directly into the pattern of inter­
connections between processing elements.
Connectionist models and artificial neural systems are inspired by biological systems such as the brain, where knowledge is stored and processed by a large number of interconnected neurons. Neurons have an elementary computational capabilities, but they operate in a highly parallel fashion. A connectionist architecture uses connections, rather than memory cells, to store information. Each connection have a weight associated with it, and the pattern of weight forms the knowledge representation. These connections are used to connect a huge number of processing cells, each of them is capable of performing a very basic operation such as bit comparison.

Connectionist systems are inherently fault tolerant. By encoding the knowledge in the entire network rather than in particular memory locations, the system is able to withstand the loss of individual cells without significant performance degradation. Connectionist architectures are weak in the area of programmability. Furthermore, they lack the existence of an efficient scalable self-learning mechanism.

A connectionist system can either be fully or virtually implemented in hardware. In a fully implemented system, each neuron or cell is represented by a processor, and each connection is represented by a physical link. In a virtual system the entire network is divided into groups, one for each processor. Connections among these groups are multiplexed over the physical interprocessors links. Interconnections among neurons of the same group, weights and other system parameters are stored in local memories. With the current VLSI technology limitations, it is impossible to build a large fully implemented connectionist system. Most connectionist
systems are currently implemented by simulating their behaviour on a uniprocessor or a bus-based architecture using a small number of processors. Examples of connectionist systems are the Hypernet [73] and the Boltzmann machine [36].

2.6.5 INTELLIGENT INTERFACE MACHINES

The main use of these machines is for man-machines interface applications. This include, speech recognition, pattern recognition, image processing and computer vision. Intelligent interface systems are not pure symbolic processors. They may use a variety of knowledge representations. Examples of Intelligent interface machines are: Harpy and IBM Nat.Task [127] for speech recognition, PIPE [80] and Zmob [137] for pattern recognition and image processing, and finally Warp [11] and Butterfly [31] for computer vision.

2.6.6 THE COLUMBIA DADO PRODUCTION SYSTEM MACHINE

The DADO machine developed at Columbia University, is a binary tree-structured MIMD parallel processor [121] designed for a rule-driven "production system" applications. A production system is a rule-based forward-chaining expert system. The system drives forward from facts using a database of productions (rules with if-then form) to derive conclusions. A Production Memory (PM) holds the rules and a Working Memory (WM) holds the facts. The inferencing system cycles in sequence through three phases: Match, Select, and Act or Fire. (See Chapter 4 for more details).

DADO2 is a top-down design for a production system machine. This machine is the second implementation of the DADO class of machine architectures. Other implementations are DADO1 and DADO2 [120]. DADO2 incorporate 1023 PEs
based on the Intel 8751, an 8-bit microprocessor. Each PE includes about 16 Kbytes of RAM and a specialised I/O switch. It is represented by a node in the tree, and the nodes are arranged in a 10-levels binary tree. Each PE is capable of operating in either SIMD or MIMD mode. In SIMD mode, the PE executes instructions broadcast by some ancestor PE in the tree. In MIMD mode, a PE no longer receives instruction from an ancestor, and instead it executes instructions stored in its own memory. The architecture of the DADO2 is shown in Figure 2.24.

A commercial version of DADO2 with between 3 and 8191 PEs built on the 32-bit Motorola 68020 has been announced by Fifth Generation Computer Corp. There are two versions, the first is for rule-based and sorting and matching problems, and the other is for signal interpretation. This machine can be programmed in one of the following languages: Parallel C, Parallel Lisp, Parallel Fortran, and OPS5. Finally, several researchers have suggested that massive parallelism in the style of DADO2 is ineffective in speeding up the execution of production systems [57].
CHAPTER III

Introduction to Parallel Programming
3.1 Introduction

The recent advances in hardware technology and computer architecture (as discussed in Chapter 2) inevitably lead to a powerful and faster parallel computer systems. In order to fully exploit the potential parallelism, we need an efficient notation for expressing concurrent operations. Problems for parallel computer systems require some extra programming facilities, and are considerably more difficult to program than those for a conventional uniprocessor computer system since the programmer is faced with additional complex decisions to make in order to balance the problem requirements against the available resources.

Creating efficient multiprocessor software is one of the most active research and development endeavours (see Quinn [107], Wolfe [141], Stone [124]). The most obvious and sound approach would be to program a given problem directly for a parallel processing algorithm, if possible.

The two new concepts behind the recent ideas of parallel programming theory are parallelism and asynchronism of programs. Parallel programming was defined by Gill [53] as the control of two or more operations which are performed virtually simultaneously, and each of which entails following a stream of instructions. Parallel programming languages can be classified into three categories: asynchronous, synchronous and data-flow languages.

In asynchronous languages, independent operations can be applied to different parts of a program simultaneously. In this situation it is possible to specify actions which can proceed independently. Language features are required to regulate the situation where the processes wish to interact.
The asynchronous languages have been designed and implemented mainly by academics. In 1980, Ada [18,52] was developed according to the specification of the U.S. Department of Defense. Examples of asynchronous languages are: Modula-2 [139], Pascal plus [138], and Occam [47,74].

Synchronous languages are those languages in which parallel processing is applied to the data in synchronized operations. The regularity of the data enables the same operations to be applied in parallel by constraining all processes to act in harmony. Synchronous languages are used to program array and vector processors. On such processors a sequence of instructions can be overlapped on disjoint sets of data. Consequently these machines do not have the mutual exclusion or synchronization problems of an asynchronous parallel programming environment. Examples of synchronous languages are: CDC cyber FORTRAN, DAP FORTRAN, and Illiac IV CFD FORTRAN.

Data flow programming languages are recent and have been developed mainly at universities. In a data flow program it is the availability of the data which determines when an instruction is executed. If the data is available for several instructions these instructions may be executed in parallel. This is in contrast to the above languages in which the order of execution is determined by the program text and referred to as control flow. An example of data flow languages is VAL developed at MIT [1].

There are two types of parallelism namely, structured and unstructured. Structured parallelism is defined as a set of independent identical tasks that operate on different data sets. Unstructured parallelism is that resulting from having different
instruction and/or data streams. Structured parallelism is more easy to deal with than unstructured parallelism.

The first and most important step in parallelism specification process is the selection of an appropriate algorithm for the task in hand. After making the selection, the programmer has to select appropriate data structures to represent the data used by the program. For example, an array structure is more suitable than a linked list for parallel access, since we can address each of its elements independently.

The next step is to code the algorithm in an appropriate parallel language, which provides parallel constructs to allow the programmer to specify the parallelism in the program. Alternatively this task of parallelism detection can be left for the compiler to carry it out. A restructuring compiler can also be used to discover more parallelism in a program that has already been coded in a parallel language. Parallelism can be studied at different levels of a program. At the subroutine level parallelism is called coarse grain parallelism. This consists of grouping independent modules of computation in independent subroutines, which are then executed in parallel.

At the loop level, parallelism can be specified by the programmer or it can be discovered by the compiler through dependence analysis. Parallel loops are one of the most studied topics in program restructuring. Loop parallelism is classified as medium grain.

Parallelism at the basic block level can be medium or fine grain depending on the size of the block. Basic blocks can execute concurrently if they are independent. In some cases basic blocks with independences can also execute in parallel [17]. Fine
grain parallelism at the statement or operation level is also important although the resulting speed up is smaller than that at the loop level. Due to the overhead involved, fine grain parallelism is usually exploited inside each processor by utilizing different or multiple functional units, or by using pipelining.

3.2 Parallelization

There are at least three emerging parallel software design approaches based upon the concealment (or not) of the parallelism by the hardware structure. In other terms, for some architectures, the parallelism is hidden by the hardware itself whilst for others it is revealed to the user so that appropriate decisions are made as and when needed. The first approach of parallelism, is known as ‘Implicit parallelism’ which is the automatic translation of sequential programs. Implicit parallelism is normally associated with sophisticated compiling and supervisory programs, which recognize parallelism in the application program and produce programs which can be executed in parallel. This approach will be outlined in Section 3.2.1.

The second approach is the ‘Explicit parallelism’, which is used to transform a sequential algorithm into a parallel program. In this approach the programmer has to specify the tasks of the program that can be executed in parallel by means of special statements (constructs) added to the high-level programming language itself, or by coding directly in a concurrent language. Both types of languages have special constructs for tasks initiation, termination, synchronization and message passing that allow efficient coding of even more sophisticated applications.

The third approach, advocated by Backus [14] is based on the functional language model and is implemented on most Data Flow computers. The functional
approach, which is the most natural form of handling parallelism can achieve the highest degree of concurrency, since the instructions are scheduled for execution directly by the availability of their operand. However, the high cost of implementing this unstructured, low-level concurrency makes this method of less importance, at least for the present moment.

3.2.1 IMPLICIT PARALLELISM

3.2.1.1 Introduction

One of the approaches to parallelism that relies on the implicit (automatic) detection of parallel processable tasks within a sequential algorithm is the implicit approach. In this case, the compiler determines what can be executed in parallel. In other words, the implicit parallelization is performed automatically by the compiler. Several sophisticated compiling techniques were developed to automatically translate a sequential program into a form suitable for parallel processing on a particular type of machine.

One of the main factors in a program, is 'data dependency'. If a given part of a program depends on data generated by another part then the part yielding the needed data, should be executed first and not concurrently. Two parts of a program (processes) can be executed in parallel provided they do not use data generated by each other. The parallelizing compiler scans the program to detect any existing dependency relationship among the various identified processes so as to effectively schedule them for parallel execution.

However, one of the most studied detection schemes that has been given much consideration is the implicit detection of inherent parallelism within the computation of arithmetic expressions. On a uniprocessor system, the run-time of an
arithmetic expression computation is proportional to the number of operations. On
a parallel system, this run-time can be further reduced by processing many parts
of the expression in parallel (concurrently). By using the commutativity and asso­
ciativity in the expression, the height of the computational tree representation is
reduced. As an example, consider the expression:

\[(a_1 \times a_2 \times a_3 \times a_4 \times a_5 \times a_6 \times a_7 \times a_8)\]

By using the commutativity and the associativity of the ‘\times’ operator (multiplication)
the expression can be rearranged in another form (1):

(1.) \[((a_1 \times a_2) \times (a_3 \times a_4)) \times ((a_5 \times a_6) \times (a_7 \times a_8))\]

The new form (1) is suitable for parallel processing. The run-time of the
expression on a parallel computer (see Figure 3.2) is four times less than that on a
serial computer (Figure 3.1).

There are many algorithms dealing with the detection of parallelism at the
arithmetic expression level. Some of these algorithms were suggested by Baer and
Bovet [15], Kuck [82], and Wang and Liu [135].

The complexity of parallel programming increases as the organizations of par­
allel computers are becoming more and more complex. This complexity makes the
need for sophisticated software tools more urgent. Compilers are used for restruc­
turing a serial program into a parallel one. Several commercial and experimen­
tal restructuring compilers are in use including the Vast, the KAP, the PFC, and
the Parafrase restructurer developed at the University of Illinois. (see Allen [6,7],
Kennedy [79], Kuck [84,85,86]).
Figure 3.1: Binary Tree Representation of the Expression \((a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot a_5 \cdot a_6 \cdot a_7 \cdot a_8)\) for a serial computer

Figure 3.2: Binary Tree Representation of the Expression \((a_1 \cdot a_2 \cdot a_3 \cdot a_4 \cdot a_5 \cdot a_6 \cdot a_7 \cdot a_8)\) for a parallel computer
These compilers focus on program restructuring alone and they ignore other problems such as scheduling, synchronization, and communication [87]. Parafrase-2, a project being developed at the centre of supercomputing R & D of the University of Illinois, is aimed to develop a powerful parallelizing compiler that, in addition to restructuring serial programs into parallel ones, will perform communication and synchronization overhead analysis, memory management and will help the scheduling of the resulting program for execution on parallel machines. The restructurer of Paraphrase-2 is shown in Figure 3.3. Paraphrase-2 is being developed to restructure FORTRAN 77 and C programs for shared-memory parallel machines. The restructurer will also perform the first phase of scheduling called

![Figure 3.3: The Parafrase – 2 structure](image-url)
pre-scheduling. The compiler is designed to allow easy integration of the new technology with minimal effort and cost.

3.2.1.2 Program dependence

The effectiveness of automatic program restructuring depends on how accurately we can compute data dependences. In general, data dependences give us information about the way and the order, in which data are used and computed in a program. This information is then used to parallelize the program automatically. The resulting parallel program should then respect the various data and control dependences during its parallel execution.

During the execution of a program, the results of some operations, such as branches, determine whether other statements will subsequently be executed. A control dependences graph [39] summarizes those conditions that may affect a statement's execution. Control dependences represent the necessary control flow relationships that must be respected by any execution of the program, whether parallel or sequential. By using control dependence, we can eliminate unnecessary sequencing, and thus expose potential parallelism.

Graphs are a popular data structure for representing a program. Directed acyclic graphs (Dags), flow graphs and dataflow graphs are commonly used to represent programs.

Expression dags [3] are used to represent expressions which contain common subexpressions. An operator is represented by an internal node, and its children represent the operands. Thus, an edge in an expression dag expresses a 'data dependence'. A basic block dag is similar to an expression dag, but represents an entire basic block [3].
The flow graph [3] of a program is another kind of graph used to represent a program. The nodes in the graph represent computations while the edges represent flow of control. A path in the flow graph represents a possible execution sequence in the program. A flow graph expresses sequential control dependences.

In a dataflow graph, nodes represent operators and edges represent operands. All data is represented by tokens which flow along the edges of the dataflow graph. A node which has tokens on all of its input edges is ready to fire. It executes by consuming all its input tokens and producing a token on each of its output edges. Dataflow's parallelism result from the nodes's ability to fire concurrently, and the pipelining of tokens streams. A dataflow graph is the executable machine code for a dataflow machine [60].

3.2.2 EXPLICIT PARALLELISM

This parallelization approach is performed by the programmers prior to loading the program, by specifying concurrency and using special language constructs as provided by the operating system. Such a capability is usually provided by parallel computers manufacturers, which do not offer an automatically parallelizing computer. In this approach, algorithm design is far from being a simple matter and the programmer may need to change the structure of the algorithm if it is not suitable for parallel processing. Much research has been done on this approach with a particular interest to parallel constructs used for task declaration, activation, termination, synchronization, and communication. The term 'process' is used to describe a sequence of program instructions that can be performed in parallel with other program instructions. A program can therefore be represented as a number
of processes which can be executing in parallel (concurrently). If more than one process wishes to use a resource \((R)\), each process should be able to reserve and use \((R)\) without interference from the other processes; this is known as 'mutual exclusion'.

In 1965 Dijkstra [33] proposed the utilization of 'semaphores' and two new primitives \((P \text{ and } V)\) that greatly simplified the processes of synchronization and communication. A software implementation of these two primitives is the 'Test-and-Set' instruction, which is used in many systems. A key feature is the indivisible nature of 'test-and-set' instruction. Test-and-Set is a blocking synchronization primitive in the sense that its use is to block every process but one from doing something.

Other proposed synchronization and communication mechanisms are the 'conditional critical regions' [63,70,64]. In critical regions the shared variables are grouped into resources and allow exclusive access to them. Therefore, the conditional critical regions reduce potential interference and aid the establishment of invariants by grouping shared variables into disjoint sets dependent on their use. Another high-level synchronization mechanism is the 'monitor'. Monitor is a blocking synchronization mechanism unsuited to large-scale parallelism. A monitor defines a shared data structure and all operations that processes can perform on it. It functions to synchronize concurrent processes, transmit data between them, and control the order in which competing processes use shared, physical resources.

In 1965, Anderson [10] introduced five parallel constructs namely, the FORK, JOIN, TERMINATE, OBTAIN and RELEASE statements. These constructs provide the necessary information to control and implement parallel processing. The
five constructs are presented below in ALGOL-68 format:

```
FORK L_1, L_2 \ldots , L_n;

label: JOIN L_1, L_2, \ldots , L_n;

label: TERMINATE L_1, L_2, \ldots , L_n;

OBTAIN V_1, V_2, \ldots , V_n;

RELEASE V_1, V_2, \ldots , V_n;
```

where $L_i$ represents a label and $V_i$ represents a variable.

The 'FORK' statements generates $n$ paths, each commencing at the corresponding label. All the labels must be locally defined, and their scope is defined as block scope in which this statement is utilised. A parallel program can include many Forks at different levels, and the next sequence of paths may only be initiated when all the forked paths of the previous level have completed their executions.

The 'JOIN' statement is associated with the Fork statement and must occur with it at the same level. This construct is used to terminate and join all the parallel processes that have been forked, and a single path may follow. The path following the JOIN statement cannot be executed until all the $n$ generated paths have been terminated. Every generated path must include at the end of its code a branch operation to the JOIN statement. The FORK/JOIN technique is depicted in Figure 3.4.
Figure 3.4: The Fork/Join Technique
The 'TERMINATE' statement is used to deactivate program paths. Since the FORK/JOIN concept is functionally similar to the recursive call which are already present in ALGOL-68. So, the recursive call mechanism relies upon finding a condition that permits a normal exit from this subroutine at execution time. Thus, the TERMINATE statement is provided to de-activate this particular call.

The 'OBTAIN' statement is used to provide an exclusive access of the listed variables \((v_i's)\) by only one parallel path from the whole program. It locks out other parallel paths, from the use of these variables, so that to avoid mutual interference. All the variables used with the OBTAIN statement must be defined in higher level blocks.

The 'RELEASE' statement permits access to those variables that have been locked out by an OBTAIN statement. It releases all those no-longer required shared variables. The OBTAIN/RELEASE concept is one approach which is implemented to solve the shared variables interference effect in the synchronisation problem.

Finally, an ever-present danger accompanying the use of mutual exclusion is 'deadlock', in which processes are waiting for events that cannot occur. For example, assume two processes attempt to acquire exclusive access to two shared variables but in reverse order to each other. Each is preventing the other to proceed since it is holding a variable requested by the other. Many techniques have been devised to avoid deadlocks and to recover from them. For example, deadlock cannot occur if all tasks obtain locks in the same order.

In the next section we shall describe the parallel programming on the Sequent Balance 8000 system.
3.3 Elements of Parallel Programming for The Balance 8000

The two most important ideas behind the design and use of multiprocessor systems are increased performance and reliability. When designing programs for multiprocessor systems, it is necessary to distinguish between two ideas; the adaptation to parallel environments of large tasks already encountered in uniprocessor, and the creation of new algorithm specific to a particular architecture. With an existing sequential program, some modelling process can be applied to determine which parts of the original program can be partitioned into concurrent tasks and executed in parallel. In adapting an application for multitasking, the two goals are:

(I) Run as much of the program in parallel as possible.

(II) Balance the computational load as evenly as possible among parallel processes.

Sequent systems support the two basic kinds of parallel programming: 'multiprogramming' and 'multitasking'.

Multiprogramming is an operating system feature that allows multiple different programs to be executed concurrently, on a computer. A multi-user operating system is a good example of this. DYNIX, the operating system used in Sequent Balance systems, (see Chapter 2), is an enhanced UNIX operating system. In normal versions of the Unix operating system, executable processes wait in a run queue; when the CPU suspends or terminates the execution of one process, it switches to the process at the head of the run queue. Dynix uses the same technique, except that a pool of processors is available to execute processes from the run queue. Dynix balances the system load amongst the available processors, keeping all processors
busy as long as there is enough work available. Thus, each processor is used at its full computing capability.

Multitasking is a programming technique, in which a single application can be divided into multiple processes executing concurrently. The Sequent system supports multitasking by allowing a single application to consist of multiple, closely cooperating processes, i.e. through shared memory. The sequent language software includes multitasking extensions to C, Pascal, and FORTRAN. The Dynix parallel programming library includes routines to create, synchronize, and terminate parallel processes from C, Pascal and FORTRAN programs.

3.3.1 PROGRAMMING METHODS

The first step in adapting an application for efficient multitasking is to choose the right programming method. There are two main multitasking programming methods: data partitioning and function partitioning.

Most application naturally lend themselves to one of these two methods. The two programming methods are described in the following subsections (3.3.1.1 and 3.3.1.2).

3.3.1.1 Data partitioning

This method involves creating multiple, identical processes and assigning a portion of the data to each process. Data partitioning is appropriate for applications that perform the same operations repeatedly on large collections of data. Applications that require loops to perform certain operations on arrays or matrices, are well suited for this method. Data partitioning is done by creating multiple, independent and identical processes to execute the loop iterations in parallel. The
The data-partitioning method described below is sometimes called microtasking, and it has the following characteristics:

- All parallel processes share some data and create their own private copies of other data.
- The division of the computing load adjusts automatically to the number of available processes.
- Tools, specially designed for data partitioning, are used by the program to control data flow and synchronisation.

The programmer determines which data is shared by the parallel processes, and how the program adjusts to the number of available processors. The DYNIX Parallel Programming Library includes three sets of routines: a microtasking library (see Table 3.1), a set of routines for general use with data partitioning programs (see Table 3.2), and a set of routines for memory allocation in data partitioning programs (see Table 3.3).

In the remainder of this section we are going to explain how to structure a microtasking program:

1. Each loop to be executed in parallel is contained in a subprogram. This subprogram should handle the following tasks:
   
   (i) Scheduling, which is to determine the loop iterations to be executed by each process.

   (ii) Protecting code sections that contain dependent variables, in order to guarantee correct results. The subprogram may contain calls to some Parallel Programming Library functions that synchronize the parallel processes at critical points by using locks, barriers and other semaphores.
<table>
<thead>
<tr>
<th>ROUTINES</th>
<th>DESCRIPTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>m-fork</td>
<td>Execute a subprogram in parallel</td>
</tr>
<tr>
<td>m-get-myid</td>
<td>Return process identification number</td>
</tr>
<tr>
<td>m-get-numprocs</td>
<td>Return number of child processes</td>
</tr>
<tr>
<td>m-kill-procs</td>
<td>Terminate child process</td>
</tr>
<tr>
<td>m-lock</td>
<td>Lock a lock</td>
</tr>
<tr>
<td>m-multi</td>
<td>End single-process code section</td>
</tr>
<tr>
<td>m-next</td>
<td>Increment global counter</td>
</tr>
<tr>
<td>m-pause-procs</td>
<td>Suspend child process execution</td>
</tr>
<tr>
<td>m-release-procs</td>
<td>Resume child process execution</td>
</tr>
<tr>
<td>m-set-procs</td>
<td>Set number of child processes</td>
</tr>
<tr>
<td>m-single</td>
<td>Begin single-process code section</td>
</tr>
<tr>
<td>m-check</td>
<td>Check in at barrier</td>
</tr>
<tr>
<td>m-unlock</td>
<td>Unlock a lock</td>
</tr>
</tbody>
</table>

Table 3.1: Parallel Programming Library Microtasking Routines

(iii) Synchronizing processes as necessary.

(iv) Handling I/O, if required by the loop.

The subprogram can be set up to use either static scheduling or dynamic scheduling. This is described in section 3.3.4.

(II) The calling program handles the following tasks:

(i) Determine the number of processes to be used to execute the loop. The maximum number of processes that the system will allow the program to create is the number of CPUs on-line minus one. The program can either
### Table 3.2: Parallel programming Library Data-Partitioning Routines

<table>
<thead>
<tr>
<th>ROUTINES</th>
<th>DESCRIPTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpus-online</td>
<td>Return number of CPUs on-line</td>
</tr>
<tr>
<td>s-init-barrier</td>
<td>Initialize a barrier</td>
</tr>
<tr>
<td>s-init-lock</td>
<td>Initialize a lock</td>
</tr>
<tr>
<td>s-lock or s-clock</td>
<td>Lock a lock</td>
</tr>
<tr>
<td>s-LOCK</td>
<td>C macro for s-lock</td>
</tr>
<tr>
<td>s-unlock</td>
<td>Unlock a lock</td>
</tr>
<tr>
<td>s-UNLOCK</td>
<td>C macro for s-unlock</td>
</tr>
<tr>
<td>s-wait-barrier</td>
<td>Wait at a barrier</td>
</tr>
</tbody>
</table>

### Table 3.3: Parallel Programming Library Memory-Allocation Routines

<table>
<thead>
<tr>
<th>ROUTINES</th>
<th>DESCRIPTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>bk or sbk</td>
<td>Change private data segment size</td>
</tr>
<tr>
<td>shbk or shsbk</td>
<td>Change shared data segment size</td>
</tr>
<tr>
<td>shfree</td>
<td>Do-allocate shared data memory</td>
</tr>
<tr>
<td>shmalloc</td>
<td>Allocate shared data memory</td>
</tr>
</tbody>
</table>

call the Parallel Programming Library routine `m-set-procs`, or use the default number of processes set by the system. That is the number of CPUs on-line divided by two (CPUs on-line/2).
(ii) Calling the Parallel Programming Library routine \texttt{m-fork} which forks a set of child processes and assigns an identical copy of the subprogram to each process for parallel execution. \texttt{m-fork} creates a copy of any private data for each process.

(iii) Suspending or terminating parallel processes. When all the loop iteration have been executed, control returns from the subprogram. At this point, the calling program can perform one of the following operations:

1. Terminates the parallel processes, i.e. the child processes created by the previous call to \texttt{m-fork}. This is done by calling the parallel library routine \texttt{m-kill-proc}.

2. Suspends the execution of processes, created by the \texttt{m-fork} call, until they are needed for another subprogram. This is done by calling the parallel library routine \texttt{m-park-procs}.

3. Leaves the parallel processes to spin in a "busy wait" state until they are used again.

3.3.1.2 Function partitioning

This programming method involves creating multiple unique processes and having them simultaneously perform different operations on a shared data set. The processes may be created within a single program or they may be independent programs created at the operating system level.

At the program level, function partitioning is different than simple multiprogramming since the independent processes need to cooperate to solve a single application. There are many methods and applications for function partitioning.
Applications such as flight simulation, program compilation and control process are well adapted. The two basic techniques used for function partitioning are: the fork-join technique and the pipeline technique. By using one or a combination of these techniques, you can solve any function partitioning application.

The fork-join construct is appropriate for applications in which no major function requires the results of any other. Each major function must be independent of the others. The algorithm for fork-join application is:

1. Fork a set of parallel processes; that is, create a set of processes, giving each one access to certain shared data.
2. Assign a task to process.
3. Protect any data that could be accessed simultaneously by two or more processes.
4. Join all the parallel processes. Each process finishes its task and wait at a barrier until all the other processes are finished.
5. Proceed with serial execution for the following code.

The pipeline technique is appropriate for applications in which the major functions are dependent on each other and the size of data is very large. The algorithm for a pipelined application is:

1. Create a set of child processes, giving each one access to a set of shared data.
2. Assign a task to each process.
3. The first process performs calculations on a portion of the data, write the results to shared memory, and notifies the next process that the results are available for processing.
(4) Give the process new data sets and make each subsequent process use the results of the previous process until all the work is done.

(5) Terminate the processes or, if there are other tasks to be done, leave them to spin until they are assigned new tasks.

The Dynix parallel programming library contains several routines that can be used for function-partitioning applications. Some of these routines are described below: The fork() system call creates a duplicate copy of the current process. The parent process sets up a shared memory region and one or more locks, then forks one or more child processes and assigns them some tasks. The number of processes to create is determined by the system call tmp-ctl(). This call determines how many processors are available for use. Child processes inherit the parent's complete memory image, including access to shared memory and locks. They are identical to the parent process and they can be designed to choose their own tasks based on the order of their creation.

The parallel programming library routines s-init-barrier and s-wait-barrier initialize a barrier and cause processes to spin until all related processes arrived at the synchronisation point. Another way of synchronisation is the use of signals. This is done by system calls and library routines such as sigpause(), sigvec(), sigblock(), signal() and Kill(). However, on a sequent system many process synchronisation tasks can be performed more easily and efficiently using locks and shared variables.

3.3.2 PROCESS CREATION AND TERMINATION

In Dynix, as in other UNIX-based operating systems, a new process is created by using a system call called a fork. The m-fork() routine in the parallel programming library does exactly this. The m-fork() routine forks a set of child processes
and assigns an identical copy of the subprogram to each process for parallel execution. The child processes are duplicated copies of the process which created it (parent), with the same data, register content and program counter. The child process can access any open file or shared memory, that the parent has access to. After all the child processes have executed the subprogram, the child processes can either be terminated by a call to the routine m-kill-procs or suspended by a call to the routine m-park-procs, or just left to spin until they are re-used by another m-fork() call.

The number of child processes used by the m-fork() call can be set with a previous call to m-set-procs. This number can be set as low as one or as high as the number of CPUs on-line minus one. If m-set-procs has not been called, the number of child process defaults to the number of CPUs on-line divided by two (CPUs on-line/2).

Each child process is given a private integer number called an identification number (ID), which identifies it within the set of child processes being created. The parent process's ID is always zero. Child process's are assigned ID's in the order of their creation: the first child process's ID is 1, the second's is 2 and so on. A process's ID is determined by a call to m-get-myid routine.

There are two points to be considered when using m-fork, the first is that fork operation is relatively expensive, therefore a parallel application forks as many processes as it is likely to need at the beginning of the calling program, and does not terminate any process until the program is complete. The second point is that it is necessary to fork enough processes to execute a program quickly, but not so
many that they are often sitting idle and consuming CPU cycles, waiting for work.

The parallel programming library includes three routines to manage child processes after m-fork() calls: m-park-procs, m-rele-procs and m-kill-procs. The m-park-procs routine suspends the execution of child processes created by an m-fork call. Typically, if the program requires much computation before the next m-fork call, it can suspend the child processes and relinquish their CPUs to be used by other processes. The m-rele-procs routine resumes child process execution when the child processes are again required. After the last m-fork call, the program should call the routine m-kill-procs to terminate the child processes.

3.3.3 CREATION OF SHARED AND PRIVATE DATA

Typically, multi-tasking programs includes both shared and private data. 

shared data is accessible by both parent and child processes. Private data is accessible by only one process. There are several advantages to sharing data. Firstly, it uses less memory than having multiple copies. Secondly, it avoids the overhead of making copies of the data for each process. Finally, it provides a simple and efficient way for communication between processes. If the program includes any shared data, the process's virtual memory space also contains a shared data area and a shared heap. Figure 3.5 illustrates the virtual memory contents of a process.

3.3.4 TASKS SCHEDULING

Tasks can be scheduled among processes using three types of algorithms: prescheduling, static scheduling, or dynamic scheduling.

In prescheduling, the programmer has to determine the task division before the compilation of the program. Prescheduled programs cannot automatically
Figure 3.5: Process Virtual Memory Contents
balance the computing load according to the data or the number of CPUs in the system. Therefore this method is appropriate only for function partitioning applications, where each process is performing a different task.

In static scheduling, the tasks are scheduled by the processes at run time, but they are divided in some predetermined way. Static scheduling requires no communication between processes. In data-partitioning program, static scheduling is used to divide the loop iterations evenly among the processes. The computing time for each iteration of the loop should be approximately the same. The static scheduling algorithm for a process is:

1. Figure out which tasks I will do.
2. Do all my tasks.
3. Wait until all other processes finish their tasks.

The algorithm involves the following steps:

1. Call the PPL routine m-get-numprocs to determine the number (M) of processes created by the m-fork call.
2. Call the PPL routine m-get-myperid to determine the ID (N) of the process.
3. Start by executing the $N^{th}$ loop iteration.
4. Execute every $M^{th}$ iteration until reaching the end of the loop.

Static scheduling produces static load balancing. Since the partition of the tasks is statically determined, several processes may stand idle while one process completes its share of the jobs.

In dynamic scheduling, each process schedules its own tasks at run time by checking a task queue or a "do-me-next" array index. The scheduling algorithm for a process is:
1. Wait until some tasks appear in the queue.
2. Remove the first task from the queue and process it.
3. IF there are more tasks in the queue Go To step 2 ELSE Go To step 1.

The algorithm includes the following steps:
1. Check shared loop index and verify that there is still work to be done.
2. Increment the shared loop index by N. This is done by using a PPL routine called $m$-$next$ which automatically increments a global counter.
3. Execute N iterations.
4. Repeat steps 2 through 3 until all the work is finished.

Dynamic scheduling provides dynamic load balancing. All processes are kept busy as long as there is work to be done. However this method creates more communication overhead than static scheduling, since all the processes have to access a single shared task queue.

### 3.3.5 TASKS SYNCHRONIZATION AND MUTUAL EXCLUSION

In order for processes to work together, they must have the ability to communicate and synchronize. In the Sequent Balance system this communication is achieved only through shared memory.

Synchronization has two uses: the first is to constrain the ordering of events and the second is to control interference.

For example, consider a parallel algorithm in which one process (process 1) writes into a buffer and another process (process 2) is reading from it. Obviously process 1 cannot write into a full buffer, and process 2 cannot read from an empty
buffer. The mechanism to prevent these events from happening is an example of the first kind of synchronization.

The second use of synchronization is to control interference. This can be set up by using mechanisms called semaphores and locks. A lock has two values: **locked** and **unlocked**, which ensure that only one process at a time can access a shared data structure. A process, which attempts to access a shared data, has to wait until the lock associated with the data structure is unlocked. The process then locks the lock, accesses the data structure, and unlocks the lock. Figure 3.6 illustrates how a lock is used to prevent multiple processes from executing a dependent section simultaneously.

![Diagram of lock usage](image)

**Figure 3.6: Role of a lock in protecting the critical region**

While a process is waiting for a lock to become unlocked, it spins in a tight loop, producing no work, hence the name “spinlock”. This spinning is also referred to as **busy wait**.
A semaphore is a shared data structure used to synchronize the actions of multiple cooperating processes.

The parallel library contains the following routines for using the locking mechanisms: \texttt{m-lock}, \texttt{m-unlock}, \texttt{s-init-lock}, \texttt{s-lock} and \texttt{s-unlock}.

The \texttt{m-lock} and \texttt{m-unlock} routine support only one lock per looping subprogram. They are easier to use than other locks because, they do not require the declaration of the lock before using it. \texttt{m-lock} and \texttt{m-unlock} are faster than other locks, since they do not pass the lock address as an argument. If the program requires more than one lock, the \texttt{s-init-lock}, \texttt{s-lock} and \texttt{s-unlock} routines can be used.

The following shows how these routines can set up two critical regions:

\begin{verbatim}
:::
\texttt{s-init-lock(a)};
\texttt{s-init-lock(b)};
:::
\texttt{begin}
:::
\texttt{\ldots \ldots\ldots }
\texttt{\ldots\ldots\ldots }
\texttt{\ldots\ldots\ldots }
\texttt{\ldots\ldots\ldots }
\texttt{s-lock(a);}
\texttt{\{critical region 1\}}
\texttt{s-unlock(a);}
:::
\texttt{s-lock(b);}
\texttt{\{critical region 2\}}
\texttt{s-unlock(b);}
:::
\texttt{end;}
\end{verbatim}
A parallel subprogram sometimes contains a section which depends on all the processes having completed execution of the preceding code. In such situations, some sort of synchronization such as barriers is required to synchronize the processes. The parallel library contains routines to set up two kinds of barriers. The routine 'm-sync' synchronizes all the processes at a single, pre-initialized barrier. To set up more than one barrier, the subprogram can call 's-init-barrier' to initialize a barrier and then call 's-wait-barrier' to synchronize processes at the barrier. 's-wait-barrier' delays the process in a busy wait until all the processes have called 's-wait-barrier' (waiting at the barrier). At that point all processes exit the busy wait simultaneously.

Some program operations depend on previous operations, while some may be executed in any order. There are two classes of program dependences: data dependence and control dependence. There are three types of data dependences. The first one is called flow dependence. It occurs when one operation set a data value that is used by a subsequent operation. As an example consider the following three statements:

(Example 1)

1. \( X = Y + Z \)
2. \( D = X \times 4 \)
3. \( E = X + 2 \)

As we can see, the value of \( X \) is set in statement 1 and used in statement 2. Statement 3 also depends on statement 1.

The second type of data dependence is called antidependence. It occurs when one operation uses a memory location that is loaded by a subsequent operation. In
the following example, (Example 2) statement 1 must execute before statement 2, since the current value of \( Y \) is used by statement 1.

(Example 2)

1. \( X = Z + Y \)
2. \( Y = Z \times 2 \)
3. \( X = E + 4 \)

The last type of data dependence is output dependence, and occurs when one operation loads a memory location which is also loaded in a subsequent operation. In Example 2 statement 3 must execute after statement 1, otherwise the variable \( X \) will contain the wrong data value at the end of this program segment.

The other class of program dependence is the control dependence. This class includes all the dependences that result from the required flow of control in a program. As an example consider the following program segment:

(Example 3)

\[
\begin{align*}
& s_1 : \text{IF RESULT} > \text{LIMIT THEN} \\
& s_2 : \quad \text{RESULT} = \text{LIMIT} \\
& s_3 : \quad \text{ELSE RESULT} = \text{SUM} + 4
\end{align*}
\]

The control dependence graph for Example 3 is shown in Figure 3.7. In Example 3 statements 2 and 3 are conditionally executed depending on the results of the test in statement 1.

Figure 3.7: Control dependence graph for Example 3
As example of data dependence consider the five following statements of a program:

(Example 4)
\[ s_1 : \quad X = Y + Z \]
\[ s_2 : \quad Y = Z + 3 \]
\[ s_3 : \quad W = X + Y \]
\[ s_4 : \quad \text{IF } U > 10 \text{ THEN} \]
\[ s_5 : \quad Y = U * 2 \]

The data dependence graph for these statements is shown in Figure 3.8.

![Data dependence graph for Example 4](image)

In Figure 3.8 statement \( s_3 \) is data-dependent on statements \( s_1 \) and \( s_2 \). Thus, the execution of \( s_3 \) must follow the execution of \( s_1 \) and \( s_2 \). Consequently, the execution of \( s_1, s_2 \) and \( s_3 \) cannot be performed concurrently.

### 3.4 Parallel Algorithms for Multiprocessors

Efficient serial algorithms do not necessarily lead to efficient parallel algorithms, because of the structure of the computer itself. Research in parallel computation...
shows that often the techniques used in serial algorithms cannot be directly applied to parallel computers. Hence the need to find new techniques which adapt more to parallel computers and which lead to designing more efficient parallel algorithms. Since the late 1960’s extensive research has been performed on SIMD algorithms, and many algorithms have been developed for these computers. The area of MIMD algorithms has only recently been investigated. There are still few results available concerning the specification, design and analysis of MIMD multiprocessor algorithms. This basically results from the technical problems associated with MIMD systems.

A parallel algorithm for a multiprocessor is a set of \( m \) (\( m \geq 1 \)) concurrent processes that may operate simultaneously and cooperatively to solve a given problem.

Hence, in the execution of a process there may be some points where it can communicate with other processes. These points are called “interaction points”. These points divide a process up into stages, and a process may communicate with other processes at the end of each stage.

Parallel algorithms for multiprocessor systems were classified into synchronized and asynchronous algorithms [88]. The two classes are described in the following two subsections.

Finally, there are two methods of decomposing a multiprocessor algorithm into a set of processes: static decomposition and dynamic decomposition. In static decomposition, the set of processes as well as any precedence relationship amongst them are known before execution. In this method the amount of data communication is kept very low, provided the number of processes is small. In
dynamic decomposition, the set of processes changes during execution. Dynamic decomposition algorithms create more process communication overhead, but the computing load can be very evenly distributed because no process is idle while there is still work to be done.

3.4.1 SYNCHRONIZED PARALLEL ALGORITHMS

A synchronized parallel algorithm is a parallel algorithm in which some processes are forced to wait for the required input. At the "interaction points" the next process is not achieved until all the required input is available, i.e. some stages of a process are not activated until another process has finished a certain portion of its program.

As an example, consider constructing a parallel algorithm, using $p$ processes to compute the following multiplication:

$$M = c(1) \times c(2) \times \ldots \times c(n)$$

A synchronized parallel algorithm may be constructed by partitioning the above expression into $p$ subsets. Thus,

$$M = \left( c(1) \times \ldots \times c(k) \right) \times \left( c(k+1) \times \ldots \times c(2k) \right) \times \ldots \times \left( c(p-1)k+1 \times \ldots \times c(n) \right).$$

where $k = \left\lfloor n/p \right\rfloor$.

The parallel algorithm assigns to each process the job to perform the multiplication of one subset. This is done in the following way:
process 1 → task 1 = c(1) *...* c(k)
process 2 → task 2 = c(k + 1) *...* c(2k)

process p → task p = c((p - 1)k + 1) *...* c(n)

After the p processes complete their tasks the p results are multiplied to give the final result. This is done using one process. It is clear that to do the last multiplication, it is necessary to have the results of all the processes available. Otherwise, a wait is imposed until all the processes are ready.

The weakness of a synchronized algorithm is that all the processes that have to synchronize at a given point wait for the slowest amongst them. Some processes may be blocked at a given time, so the performance of the algorithm is degraded.

3.4.2 ASYNCHRONOUS PARALLEL ALGORITHMS

In asynchronous parallel algorithm there is always a set of global variables accessible to all processes. When a stage of a process is completed, the process first modifies the shared variables, and then activates the next stage or terminates itself. The communication between processes is achieved through the global variables or shared data. The processes are only blocked by the use of critical regions, to ensure reliable updating.

The asynchronous parallel algorithms are in general more efficient than the synchronized ones, for many reasons, some of these reasons are:

1. In asynchronous parallel algorithms, processes never wait for inputs at any time but continue execution or terminate according to whatever information is currently contained in the global variables.
2. Asynchronous parallel algorithms will adapt to make maximum use of the parallelism during most computation times, and processes finish about the same time.

3. Asynchronous parallel algorithms are more reliable, because if some processes are blocked forever, the algorithm will still be able to calculate the solution of its problem, as long as just one process is still running (not blocked).

3.4.3 PERFORMANCE OF PARALLEL ALGORITHMS

In the study of parallel computer algorithm, we need some measure of efficiency to evaluate the algorithm. The speed up ratio and efficiency of a particular parallel algorithm are the most important characteristics to show the performance of a parallel algorithm. To study this performance we need to measure the computation time. This is usually proportional to the total number of arithmetic operations in the program. For programs with little arithmetic, the computational time may be proportional to the number of memory accesses, or the number of I/O transmissions.

Let $T_p$ denote the computation time on a parallel computer with $p$ processors, and $T_1$ denote the computation time of either, the equivalent sequential algorithm measured without any of the parallel constructs and with all the data in local memory, or the parallel algorithm running on one processor.

The speed up ($s_p$) is defined as:

$$s_p = T_1 / T_p \geq 1$$

In the former case (sequential algorithm), the speed-up measures how much faster is the parallel version compared to the sequential. In the case of parallel algorithm
running on one processor, the speed-up measures the degree of parallelism exhibited by the parallel algorithm.

When no optimal sequential algorithm is known for solving a problem, the efficiency of a parallel algorithm for that problem is used to evaluate its cost. This is defined as follows:

\[ E_p = \frac{s_p}{p} \leq 1 \]

The longer the processors are idle, or carry out extra overheads, which are introduced through parallelization of the problem, the smaller will be the efficiency \( E_p \).

Obviously, it is apparent that a multiprocessor system with \( p \) identical processors cannot execute a parallel program more than \( p \) times faster than a single processor. The maximum speed-up expected will be \( p \). The speed-up is limited by factors introduced by the communication, synchronization and data exchange amongst all the processes. On the other hand, since a job is subdivided into \( p \), or possibly more, individual subtasks causes an additional overhead explicitly associated with a multiprocessor system. This task partitioning may create three types of overheads:

1. If the number of processes available is \( p \) and these are less than \( p \) subtasks remaining to be performed, then some of these processes must remain idle until the completion of all the subtasks.

2. The second type of overhead is known as "organizational" overhead. This occurs when a subtask generates results required by another subtask (as input). Thus,
a mechanism is necessary to ensure the proper sequencing of subtasks. In this case, the latter subtask has to wait until the first subtask produces the results.

3. The third type of overhead occurs when $n$ processes, where $n$ is less than the number of processes available in the system, access simultaneously a shared data. Such an overhead is associated with checking the number of simultaneous accesses not to exceed the fixed limit. Consequently, the process's time is wasted while waiting to gain access.

In summarising, two types of overheads can be distinguished: static overheads and dynamic overheads. Static overheads are due to the design of hardware and software. This includes the overheads from the subdivision of the tasks, allocation of tasks to processes, contention control by hardware and software. Dynamic overheads are due to the interference between one or more subtasks running on different processors, causing one or more of them to wait.

Static overheads can be calculated before execution and will be fixed. The dynamic ones, however, do not only depend on the algorithm, but also on the detailed timing considerations which may vary even if the same task is executed, on the same machine on consecutive runs.
CHAPTER IV

Introduction to Expert Systems
4.1 A Brief History

Barr and Feigenbaum [19] have defined Artificial Intelligence as: "the part of computer science concerned with designing intelligent computer systems, that is, systems that exhibit the characteristics we associate with intelligence in human behavior-understanding language, learning, reasoning, solving problems and so on".

In other words, the goal of AI scientists has always been to develop computer programs which are able to think and perform tasks that are presently done better by humans (reasoning, learning etc.).

Until the 1960's, the idea of a machine that "reasons" remained strictly theoretical. GPS (General Problem Solving), an AI program produced by Alan Newell and Herbet Simon, was the first program to use something resembling rules. The earliest expert system "DENDRAL", developed at Stanford University, dates back to 1965. It was a mass-spectrography program designed to isolate organic compounds. In 1971 ANL International, a company specializing in medical laboratory automation, introduced the first computer-based medical diagnostic system. Later, in 1974 a rule-based expert system (MYCIN) emerged from the Stanford Heuristic Programming Project (HPP). MYCIN was designed to diagnose certain infectious diseases and recommends appropriate drug treatment. The MYCIN experiment is widely seen as a milestone in AI research, and a substantial share of expert system literature is devoted to its design.

From the latter half of the 1970's AI scientists began to realize a substantially important point: to make a program intelligent, provide it with much high-quality
specific knowledge about some problem area. Human experts achieve outstanding performance because they are knowledgeable (see Hayes [66]).

It became clear that there were advantages attached to the strategy of representing human knowledge explicitly in pattern-directed modules, instead of encoding it into an algorithm that could be implemented using more conventional programming techniques.

This process of rendering the knowledge explicit in a piecemeal fashion seemed to be more in tune with the way that experts store and apply their knowledge on the other hand this method of programming allows for fast prototyping and incremental system development.

The Japanese launching of their Fifth Generation project, at an international conference in Tokyo in the Autumn of 1981, helped to crystallize an interest which has already been growing in the West, but placed the development targets well beyond those that most western researchers would have set for themselves. The report of the Japanese Information Processing Development Centre - the Jipdec Report (JIPDEC, 1981) stated the target in these words: "The Fifth Generation Computer Systems will be Knowledge information Processing systems having problem-solving functions of a very high level. In these systems, intelligence will be greatly improved to approach that of a human being".

The first four generations of computing were modelled exclusively on Von Neumann architecture, the conventional computer design incorporating an arithmetic logic unit and several reusable registers. In contrast, the Japanese team aims at a fifth generation of computers whose cognitive behavior would be all but indistinguishable from that of humans. The project's focus is distributed among expert
systems, natural language programs, voice and vision recognition, robotics, and supercomputers.

At this point expert system research moves away from the aims of AI projects. AI tries to obtain autonomous machine entities that will interface with, but not rely on, human cognition. In contrast, the expert system is wholly dependent on human expertise. In fact, the efficacy of the system is only as strong as the detail of the expert system builder's knowledge base.

This period (after 1975) has seen the development of a number of special-purpose computer programs, systems that were expert in some narrow problem area. These programs were called expert systems and a new field began.

At first, designing and building an expert system was considered more an artistic effort than a scientific enterprise. Now, however, the process is better understood and more clearly defined, due in part to the efforts of more than 40 AI scientists who collaborated on the seminar volume Building Expert Systems, a book that organizes the technical state of the art and describes the use of different expert systems - building techniques on a common problem (see Hayes [66]).

Today interest in practical expert systems is at its peak, and their application is very wide. Most feature several characteristics in common:

1. They rely on engineered knowledge bases derived from the observational experience of experts.

2. They use symbolic expressions of facts and relational attributes.

3. They are powered by heuristic inference engines, and reason with uncertainty.

4. They draw from human experience to solve human problems.
4.2 What is an Expert System

The field of expert systems research is still quite new and there is considerable disagreement among researchers as to the definition and direction of expert system work. An Expert System is a program that behaves like an expert in some problem domain. It is a computer software system which embodies organized knowledge concerning some specific area of human expertise, sufficient to perform as a skillful and cost-effective consultant.

Thus, it is a high-performance special-purpose system which is designed to capture the skill of an expert consultant such as a doctor of medicine, a chemist, mechanical engineer, etc.

Expert systems have to be capable of solving problems that require expert knowledge in some form. Therefore they are also called knowledge-based systems. In addition to the characteristics of expert performance, an expert system should be able to provide a high-level explanation of its result. It has to be capable, in some way, of explaining its behavior and its decisions to the user, as human experts do. An explanation capability is extremely important in domains for which there is a high cost for incomplete or inaccurate solution. In medicine, for example, no treatment plan will be accepted unless a satisfactory explanation can be produced. Therefore, expert systems have to have a friendly user-interaction capability that will make the system's reasoning transparent to the user.

An additional feature that is often required of an expert system is the ability to deal with uncertainty and incompleteness. Information about the problem to be solved can be incomplete or unreliable where relations in the problem domain can be
approximate. For example, we may not be quite sure that some symptom is present in the patient, or that some measurement data is absolutely correct; some drug “may” cause some problem, but “usually” does not. All this requires probabilistic reasoning. To build an expert system we have, in general, to develop the following functions:

- Problem-solving function capable of using domain-specific knowledge. This may require dealing with uncertainty.
- User interaction function, which includes explanation of the system’s intention and decision during and after the problem-solving process.

Note that each of these functions can be very complicated, and can depend on the domain of application and practical requirements.

4.3 Building an Expert System

The process of building an expert system is often called knowledge engineering, and the expert-system builder is called a knowledge engineer. Building an expert system involves a special form of cooperation and interaction between the knowledge engineer and one or more human experts in some field. The knowledge engineer extracts the essential and necessary knowledge from human experts and builds it into the expert system. Knowledge engineering relies heavily on the study of human experts in order to develop intelligent, skilled programs. As Hayes-Roth and others point out in the book Building Expert Systems [66]:

“The central notion of intelligent problem-solving is that a system must construct its solution selectively and efficiently from a space of alternatives, when
resource-limited, the expert need to search this space selectively, with as little unfruitful activity as possible. An expert’s knowledge helps to spot useful data early, suggests promising ways to exploit them, and helps avoid low pay off efforts by pruning blind alleys as early as possible.”

4.4 Main Structure of an Expert System

All expert systems include at least three basic elements (Figure 4.1):

1. A knowledge base
2. An inference engine
3. A user interface.

The knowledge base represents what is known about a given subject at the present. The inference engine knows how to actively use the knowledge in the base. The user interface facilitates communication between the user and the system; it is convenient to view the inference engine and the user interface as one module, usually called an expert system shell, or simply a shell.

In a powerful knowledge system, you might find a natural language interface that understands and formulates conversational English. In a less powerful knowledge system, you will find a less elaborate, but still user-friendly, interface.

The knowledge base is generally separate from the part of the program which performs the reasoning, the ‘inference engine’, This means that one can make at least some changes to either module without necessarily having to alter the other. Thus we might be able to add more knowledge to the knowledge base, or tune the

102
inference engine for better performance, without modifying the code elsewhere.

![Diagram of an expert system](image)

**Figure 4.1: The elements of an expert system**

4.4.1 THE KNOWLEDGE BASE

The knowledge base contains known facts about the subject, expressed as objects, attributes and conditions. It comprises the knowledge that is specific to the domain of application, including such things as simple facts about the domain, rules that describe relations or phenomena in the domain, and possibly also methods, heuristics and ideas for solving problems in this domain.

It can be distinguished from a conventional database by its symbolic, rather than numeric or literal content. When database information is processed, it is retrieved, stored or calculated. When the content of a knowledge base is processed it is rigorously manipulated according to predefined rules of logic. Consequently, a knowledge base is expressed at a high level of abstraction, dealing with classes of objects rather than the objects themselves.

The knowledge base derives from humans either directly from the experts themselves or indirectly from their works.

4.4.2 THE INFERENCE ENGINE

The inference engine is the heart of an expert system. As its name implies, the engine's main job is to draw inferences. The inference engine is the mechanism
that applies the rules of rational logic to search the knowledge base for solution
whenever a user initiates a query. It then performs the following tasks:
1. It compares information supplied in the user's query with information in the
   knowledge base.
2. It seeks specific event-related goals or casual relationships.
3. It evaluates the relative certainty of facts, based on the respective confidence
   values associated with each fact.

Searching for goals based on propositional rules, the inference engine continues
its request through the knowledge base until it finds a feasible path to an acceptable
future event.

4.4.3 THE USER INTERFACE

The job of the user interface is to exchange information between the operator
and the inference engine. It caters for smooth communication between the user and
the system, also providing the user with an insight into the problem-solving process
carried out by the inference engine. The best user interface are those which use a
natural language interface that understands and formulates conversational English;
but obviously, these need greater demands on permanent storage and memory.

The measure of the performance of an expert system is often see how good
the user interface. No matter how efficient is the inference engine or extensive is
the knowledge base, the program is only as valuable as its ability to communicate
lucidly with those who require access to its output.
4.5 Knowledge Representation in Expert Systems

There are a standard set of knowledge representation techniques, any of which can be used alone or in conjunction with others to build expert systems. Each technique provides the program with certain benefits, such as making it more efficient, more easily understood or more easily modified. The three most widely used in current expert systems are:

1. rules (the most popular)
2. semantic nets
3. frames.

4.5.1 RULES

By far, the most popular knowledge encoding method for expert systems is in the form of rules. Production rules are usually of the form:

\[ \text{IF set of conditions THEN set of actions} \]

for example:

[1] IF there is an organism which requires therapy, and consideration has been given to the possibility of additional organism requiring therapy.

THEN compile a list of possible therapies, and determine the best therapy in this list.

When the current problem situation satisfies or matches the IF part of a rule, the action specified by the THEN part of the rule is performed.
Two popular strategies for rule ordering are **forward chaining** and **backward chaining**. The rules that would control the inference procedure are commonly called **meta-rules** and are explicitly encoded in much the same manner as the domain knowledge.

Most production systems, consist of a set of production rules, a rule interpreter that decides how and when to apply (Fire) which rules, and a 'working memory' that can hold data, goals or intermediate results. Examples of such systems are: MYCIN (in medicine), DENDRAL (in chemistry), R1 (in computer systems).

### 4.5.2 SEMANTIC NETS

This is a knowledge representation method based on network structure. A semantic net consists of points called nodes connected by links called arcs representing the relationships between the nodes. They acquired their name because they were originally employed to represent the meaning of natural expression. The associated links connecting nodes often represent domain independent concepts like is-a, part-of, has-part, sub-concept-of, and instance-of.

As a simple example, consider the statements:

1. AMAL is a Female.
2. Every Female is a Mammal.

The representation of these sentences in semantic net form is shown in Figure 4.2: because the is a relation is transitive we can infer a third statement from the net:

3. AMAL is a Mammal.

The is a and others establish a property inheritance hierarchy in the net. This
means that items lower in the net can inherit properties from items higher up in the net. This is more efficient and saves space since information about similar nodes do not have to be repeated at each node.

Examples of expert systems which use networks are: CASNET (Medicine), PROSPECTOR (Geology), INTERNIST (Medicine) etc . . .

4.5.3 FRAMES

Some expert systems are frame based (e.g. PIP: in medicine, CENTAUR: in medicine . . .). A frame is a special way of representing common concepts and situations. Marvin Minsky [96] who originated the frame idea, describes it as follows:

"A frame is a data-structure for representing a stereotyped situation, like being in a certain kind of living room, or going to a child's birthday party. Attached to each frame are several kinds of information. Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is about what to do if these expectations are not confirmed".
A frame is a structure that collects together knowledge about a particular concept and provides expectations and default knowledge about the concept. Typically, the frame is represented in the computer as a group of slots and associated values. The values may themselves be other frames.

As frame example consider the following frame representing a PERSON frame:

 FRAME: PERSON

   SEX       (MALE/FEMALE)
   NAME      (string of character)
   AGE       (integer x such that 0 ≤ x ≤ 150)

An expert system can be built by providing a decision strategy to process such a set of interconnected frames. While PIP used frames exclusively, CENTAUR mixed frame and production rules representations. The mixing of techniques has provided some very rich representations and powerful computational problem solving systems.

4.6 Types of Tasks Carried Out by Expert Systems

Most expert applications fall into one or other of the following categories:

Interpretation

Which involves the analysis of data to determine its meaning.

Design

Which involves configuring objects that satisfy the constraints of the design problem.
Diagnosis

Infering system malfunction (fault-finding) from some set of symptoms.

Prediction

Infering likely consequences of given situation.

Planning

Designing actions. It involves the creation of a plan of actions to achieve a given goal.

Monitoring

It involves the continuous analysis of signals and the invocation of actions and/or alarms as appropriate.

Debugging

Prescribing remedies for malfunctions.

Repair

Executing a plan to administer a remedy for some diagnosed problem.

Instruction

Diagnosing, debugging and repairing student behavior.

Control

Interpreting the current situation, predicting the future, diagnosing the causes of anticipated problems, formulating a remedial plan, and monitoring its execution to ensure success.

Nearly all expert tasks involve large search spaces, most involve reasoning with uncertain knowledge, and some involve reasoning with time-varying data.
4.7 Expert System Application Areas and Examples

Nowadays, Expert Systems become more and more usable in great number of different areas. We will try to give here the majority of the application areas with examples of selected expert systems:

AGRICULTURE:

PLANTcd: predicts the damage to corn due to the black cutworm.

(University of Illinois) [24].

POMME: helps farmers manage apple orchards by providing advice on how to improve the apple crop.

(Virginia Polytechnic Institute and State University) [108].

CHEMISTRY:

CONGEN: helps structural chemists determine a set of possible structures for an unknown compound.

(Stanford University, DENDRAL project) [26,27].

C-13: aids organic chemists in determining the structure of newly isolated, naturally occurring compounds.

(Stanford University) [55].

DENDRAL: infers the modular structure of unknown compounds from mass spectral and nuclear magnetic response data.

(Stanford University) [8,26].
META-DENDRAL: helps chemists determine the dependence of mass spectrometric fragmentation on substructural features.

(Stanford University) [8,26].

COMPUTER SYSTEMS:

CRIB: helps computer engineers and system maintenance locate computer hardware and software faults.

(ICL, RADC and Brunel University) [2,65].

XCON: Configures VAX 11/780 computer systems.

(Carnegie-Mellon University, DEC in Hudson, Massachusetts) [13,92,106].

YES/MVS: helps computer operators monitor and control the MVS (multiple virtual storage) operating system, the most widely used operating system in large mainframe IBM computers.

(Waston Research Centre in York town, Heights, New York) [111].

ELECTRONICS

ACE: identifies trouble spots in telephone networks and recommends appropriate repair and rehabilitative maintenance.

(Bell laboratories at Whippany, New Jersey) [133].


(Tektronix) [4].

ENGINEERING
CONPHYDE: helps chemical engineers select physical property estimation methods.
(Carnegie-Mellon University) [16].

REACTOR: assists reactor operators in the diagnosis and treatment of nuclear reactor accidents by monitoring instrument readings, such as feedwater flow and containment radiation level, looking for deviations from normal operating conditions.
(E G & G Idaho) [101].

- GEOLOGY

MUD: helps engineers maintain optimal drilling fluid properties.
(Carnegi-Mellon University, NL Baroid) [77].

PROSPECTOR: acts as a consultant to aid exploration geologists in their search for ore deposits.
(SRI International) [49].

- INFORMATION MANAGEMENT

PROJCON helps a software development project manager diagnose the project’s problems and their causes.
(Georgia Institute of Technology) [130].

- LAW

LDS: assists legal experts in settling product liability cases.
(The Rand Corporation) [136].
TAXMAN: assists in the investigation of legal reasoning and legal argument using the domain of corporate tax law.

(Rutgers University) [78].

- MANUFACTURING

IMACS: assists managers in a computer systems manufacturing environment with paperwork management, capacity planning, inventory management, and other tasks related to managing the manufacturing process.

(DE C) [103].

- MATHEMATICS

MACSYMA: is a system used for the manipulation of mathematical formulas.

(Project MAC at MIT) [90].

- MEDICINE

MYCIN: is a system which is capable of diagnosing infectious diseases and of selecting antibiotic therapy for bacteremia.

(Stanford University) [25].

AI/COAG: is a system used for consultation about human hemostasis disorders.

(University of Missouri, DEC LSI-11 micros) [50,95].

- METEOROLOGY

WILLARD: helps meteorologists forecast the likelihood of severe thunder storms occurring in the central United States.

(Radian Corporation) [94].
MILITARY SCIENCE

AIRPLAN: assists air operations officers with the launch and recovery of aircraft on a carrier.

(Carnegie-Mellon University) [91].

AMVID: assists military commanders with land battle-field analysis.

(Advanced Information & Decision Systems) [35].

EXPERT NAVIGATOR: monitors navigation sensors on advanced tactical aircraft.

(Analytic Sciences Corporation) [105].

PHYSICS

GAMMA: helps nuclear physicists identify the composition of unknown substances by interpreting gamma-ray activation spectra produced when the substance is bombarded with neutrons.

(Schlumberger-Doll Research) [20].

PROCESS CONTROL

PDS: diagnoses malfunctions in machine processes by interpreting information from sensors attached to the process.

(Carnegi-Mellon University & Westinghon Electrical Corporation) [44].

SPACE TECHNOLOGY

LES: monitors the loading of liquid oxygen (LOX) for the space shuttle orbiter at Kennedy Space Centre.
Beside these areas, there are multitude of application areas in which Expert Systems researchers and designers are working and their works give birth for many new expert systems. Examples of these areas are Teaching and Tutoring (ARCHIMEDE teaching geometry, FRANCE), Business, Cartography, Portfolio Management, Statistics, Finance & Economics, Scheduling, Banking, Regulation Handling, Materials & Structures, Personnel Assessment, Pilots Aids, Infra-red Vision, Identification etc . . .

4.8 Rule-Based Expert Systems

4.8.1 INTRODUCTION

Rule-based Expert Systems (also called production systems), are those systems that use a set of IF-THEN rules (also called productions) to present the domain knowledge. Pure rule-based Expert Systems have no control structures that specify how to move between rules, although in practice most systems are given a number of additional control capabilities. A rule-based system consists of three main parts:

1. A knowledge base: which contains the known facts about the problem to be solved and the derived data or facts.

2. A rulebase: that stores the rules used by the system. A rule has the following form:

   \[ \text{IF } C_1 \land C_2 \land C_3 \land \ldots \land C_n \ \text{THEN } A_1, A_2, \ldots, A_m \]

   where \( C_{i=1,n} \) are the conditions of the rule (also known as antecedents or premises), and the \( A_{i=1,m} \) are the actions of the rule (also known as conclusions). The \( C_i \)'s form the Left-Hand Side (LHS) of the rule, while the \( A_i \)'s
constitute the Right-Hand Side (RHS). The conditions are matched against the facts in the Knowledge Base (KB), and if they are all satisfied, the rule can be applied. Thus, a rule is called 'applicable' if all its conditions are satisfied by facts in the KB. An applicable rule can then be 'fired', which means it is to perform the actions in its RHS. Firing a rule can add assertions to, or delete assertions from the KB.

3. A control strategy or inference engine: It controls the whole operation of the system. The inference engine applies the knowledge to the solution of actual problems. It is the interpreter for the rulebase. In a Rule-based system, the inference engine is maintained separated from the other two modules, the rulebase and the knowledge base. This separation is important for several reasons:

(i) It makes it possible to represent the knowledge in a more natural way. (IF-THEN rules) instead of encoding this knowledge in lower-level computer code.

(ii) It allows changes to be made in one part of the rulebase without creating side effects in other parts of the program code.

(iii) It allows the same system to be used in a variety of applications. Since, to use the system for a new application, all what a programmer needs to do is to create a new rulebase appropriate to this application.

4.8.2 STRATEGIES FOR STATE SPACE SEARCH

4.8.2.1 Forward-chaining and Backward-chaining Mechanisms

Rule-based Expert Systems use three reasoning mechanisms in their search for a solution. These are: Forward-chaining (also called data-driven), Backward-
chaining (also called goal-driven), and a mixture of both forward and backward chaining.

**Forward Chaining** works from known facts towards the goals to achieve. In its simplest form, Forward-chaining is an interactive program that performs a loop of substitution. It steps through the rulebase until it finds a rule in which the premises (conditions) match the facts or situation. Thus, the system chains forward from the conditions of the rule toward the conclusion. If an applicable rule is found, it will be used or ‘fired’ and insert new facts. Once the rule is used, it will not be used again in the same search; however the facts which is concluded as the result of the rule’s firing will be added to the knowledge base. This cycle of finding an applicable rule, firing it and adding the conclusion to the KB will be repeated until no more matched rule can be found, or a goal is reached.

**Backward chaining** or goal-directed inference which starts with the desired goal and attempts to find evidence for this to be the case. So, it works from goals to subgoals, and then eventually to known facts. The system chains backward from conclusions in the right-hand side (RHS) of a rule towards its conditions part (the LHS), trying to test if the conditions are satisfied or not. A condition is satisfied if it exists in the data base or, if taken as subgoal, is achieved by the system. Thus given a goal, the system finds a rule whose conclusion part matches the goal. The conditions of this rule then become the new goals, or subgoals, for the search. This process is then repeated on the subgoals until they are satisfied by the facts in the current state.
Backward chaining arises typically in situations where the quantity of data is potentially very large and where some specific characteristic of the system under consideration is of interest. Most typical are various problems of diagnosis, such as medical diagnosis or fault finding in electrical or mechanical equipment.

The last strategy of search is a mixture of the two strategies mentioned above. In this method, the reasoning can be conducted in one of the two following ways:

1. Given some initial assumption, we infer a conclusion by reasoning forward, and then apply backward chaining to prove these conclusions.
2. Given a starting goal, the system chains backward to some plausible reason and then drives forward to exploit the consequences of this new data. This way of reasoning is often called "backward reasoning with opportunistic forward chaining", because the forward search exploits the consequences of data as they become available, opportunistically.

4.8.2.2 Depth-First and Breadth-First Search

Forward chaining and backward chaining can be represented as searching through a branching network or tree. Trees may be searched in a number of ways. These are conveniently divided into a 'blind search' and 'informed search'. The latter is often called heuristic or intelligent search. The two basic methods of blind search are called 'depth-first' and 'breadth-first' search.

In addition to specifying a search direction (goal-driven or data-driven), a search algorithm determines the order in which states are examined in the tree or...
In this section we will present two possibilities: depth-first and breadth-first search.

In both forward and backward chaining, there is always a starting point (either a goal or a fact). This starting point is called the "root node". There are choices (branch nodes) after the starting point and more sub-branches at each branch as the matching and substitution process goes along. This decision process is called tree because every branch has a unique parent with only one exception (the root node).

Consider the graph represented in Figure 4.3. In this graph states are labelled (A, B, C, ...).

In depth-first search, when a node (state) is examined, all of its children and their descendants are examined before any of its siblings. The algorithm for a depth-first search of the graph in Figure 4.3 is:

(i) Start with X and go down to A, B, C .... Only the left-most child of each node is examined.

(ii) If the node is not the desired one then go down to the next level and choose the left-most child of that node (always moving downward).

(iii) If the search reaches the bottom level without finding the desired choice, the process returns to the last node that contained a choice and the downward motion is repeated.

Suppose that in Figure 4.3 the target node for the search is V. A depth-first search will make a great effort to reach the node V, since the process has to go through almost all nodes, down from X to C, returning from C to B and
moving down again to $D$, returning to $A$, descending to $E$ and $F$, and so on until it gets to $V$.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{depth_first_breadth_first_search.png}
\caption{Graph for depth-first and breadth-first search examples.}
\end{figure}

In contrast, \textbf{Breadth-first} search, explores the space in a level-by-level fashion. Only when there are no more states to be explored at a given level does the search move on to the next. Thus, the process examines all the nodes on the same level one by one, and if the target node is not found, then the process looks at those nodes on the next level. Consider Figure 4.3 again, and suppose that $V$ is the target node. Because $V$ is on the third level, the breadth-first search process will need to examine only 10 nodes to search $V$, in comparison with the 22 nodes in the depth-first search.

Whether the breadth-first search is more efficient than the depth-first search depends on the position in the graph of the target node.
Breadth-first search has one striking advantage over the depth-first method. It guarantees that the shortest solution, if it exists, is found. On the other hand, breadth-first search in large solution spaces can involve huge computational costs. If an optimal solution is required, then breadth-first search will tend to be better.

Finally, since we do not know in advance how deep the search may go, to find a failure or dead end before backtracking, it is sometimes convenient to place a restriction on the maximum depth of any one exploration. This is called the "depth bound". The depth bound forces a failure on a search path once it gets below a certain level.

4.8.2.3 AND/OR Tree

An AND/OR tree is a useful device for representing the behaviour of a rule-based expert system that works by problem decomposition, i.e. by decomposing high-level goals into a series of subgoals, each of which may have their own associated subgoals, and so on. Thus, to achieve high-level goals, these subgoals should be achieved first.

The nodes of an AND/OR tree are used to denote goals and subgoals, the arcs relate subgoals to their original goal. The arcs of a node can be related to specify whether all the associated subgoals have to be satisfied (AND node), or whether satisfying just one of the subgoals is enough (OR node).

As example, consider a rule $r_i$ which has the following form:

$$r_i: \text{IF } C_1 \& C_2 \text{ THEN } A$$
This will be represented by the AND/OR graph of Figure 4.4. In this case, both the condition $C_1$ and $C_2$ must be satisfied in order for the action $A$ to be performed.

![Figure 4.4: AND node](image)

In Figure 4.4, the link connecting the arcs reflects the idea that both $C_1$ and $C_2$ must be satisfied in order to achieve $A$.

In expressions of the form $C_3 \lor C_4 \rightarrow B$ the satisfaction of either $C_3$ or $C_4$ is sufficient to prove $B$. This is represented by the AND/OR graph of Figure 4.5.

In this case, arcs from $C_3$ and $C_4$ to $B$ are not connected. This reflects the idea that any of the conditions is independently sufficient to determine the truth of the conclusion.

In Figure 4.4 the nodes labelled by $C_1$ and $C_2$ are regarded as AND nodes. The nodes labelled by $C_3$ and $C_4$ in Figure 4.5 are OR nodes.

![Figure 4.5: OR node](image)
AND/OR graphs are an important tool for describing the search spaces generated by many AI problems, including those solved by logical theorem provers and expert systems. For more information about AND/OR graphs and related search algorithms see Nilsson [102].

4.9 Background and Related Research

Parallel processing and Artificial Intelligence are two key themes which have risen to the forefront of technology in the past decade. Parallel processing is coming of age and it is time that programs are implemented with parallelism in mind so that they can take advantage of the processing capabilities of new parallel systems. There has been much research done in the area of expert systems and parallel processing. Several multiple processor architectures and parallel algorithms for expert systems have been investigated in order to provide substantial performance improvement of these systems.

First, we start by presenting some of the works that have concentrated on the parallelism available in production systems written in OPS5 [41]. The research done in this area has concentrated on parallelizing the RETE match algorithm [42], which is one of the best known approaches used in the matching of objects in production systems.

Stolfo [119] and Miranker [97] have proposed several parallel match algorithms which are especially well suited for tree structured parallel machine DADO [121]. NON-VON [68,113] is another example of a production system machine which is like DADO. Gupta [58] proposed a parallel version of the RETE algorithm which exploits parallelism at a very fine grain level to reduce the variation. He suggested
that to exploit the fine-grained parallelism, a shared-memory multiprocessor with 32-64 high performance processors is desirable. Forgy [43] reported expected effects of parallel matchings based on the evaluation of several existing production systems written in the OPS5 language. Oflezer [104] has proposed partitioning algorithms for production systems to maximise the effect of parallel matching. He suggested the use of small number of very powerful processors to exploit the small amount of parallelism in the programs, instead of using massively parallel machine such as DADO. Stolfo and Ishida [75] proposed another parallel execution model of production systems, the parallel firing mechanism, and discussed two major problems within the model: the synchronization and the decomposition problems. Smith and Middleton [118] presented a matching algorithm which exploits the fine-grained parallelism available within RETE itself by using abstract node processors to emulate the RETE network directly. Gaudiot et al. [51] explored the potential of data-flow multiprocessor systems for the efficient implementation of symbolic computations, and more specifically the RETE match algorithm. They used the MIT Tagged Token Data-flow Machine for the simulation model. Gupta et al. [59] explored the role of parallelism in speeding up the execution of OPS5 rule-based systems. They describe the dataflow-like nature of the RETE match algorithm and discuss its appropriateness for parallel implementations. Finally Sabharwal et al. [109] proposed a dedicated coarse-grained, n-ary tree multiprocessor for the parallel implementation of OPS5 systems and then presented algorithms for partitioning of rules in this architecture.
In addition to the research mentioned above, which is mainly dedicated for OPS5 production systems, there have been other work in the area of parallelization of rule-based systems written in Prolog, Lisp or other languages.

Logic-based expert systems use a logic programming language (like Prolog) that has a certain amount of logical inferential capabilities. There has been much work going on in parallel Prolog [56,112,129,140]. Certainly, it can provide for effective parallel rule-based systems. There are many ways to implement parallelism in Prolog. Cornery [29] points out the following four types: OR-Parallelism, AND-parallelism, stream-parallelism, and search-parallelism. Fu and Chuang [45] proposed an efficient OR-parallel execution model with intelligent backtracking technique for Prolog. Shen and Warren [114] described the results of an investigation of OR-parallel execution of Prolog. The investigation shows that there is significant OR-parallelism in actual programs drawn from real application areas. However some workers identify the concept of a shared global address space with physically shared memory (in the Argonne models) and consider this to be serious bottleneck demanding alternative approaches [5,28]. In addition, Singh and Genesereth [116] proposed a parallel execution model (PM) for backward chaining deductions, by using dataflow principles. PM can simultaneously exploit or-parallelism, and-parallelism and pipelining.

In addition, to the work mentioned above, there has been much research that have used different approaches not based on OPS5 or the Logic programming approach. Some of these follow: Krall and McGehearty [81] have studied the potentials for parallel execution of the inference engine of EMYCIN [25], a rule-based
expert system. They used Multilisp [61] to implement the parallel version of the backward-chaining inference engine.

Arratibel et al. [12] proposed a general parallel execution model for the SATIN multi-expert system. A simulator of the inference engine, written in pascal, was tested on a multiprocessor system and the results show that about 4 to 6 processors can effectively work in parallel. Li and Marlin [89] proposed new algorithms for the parallel implementation of rule-based expert systems. They studied the parallel execution of expert systems in both interactive applications and simultaneous input applications. The algorithms were designed for an ordinary MIMD closely coupled computer. Finally, Belloch [23] described the Concurrent Inference System (CIS), and its implementation on a massively concurrent network model of computation. The rules are compiled into a large static concurrent network of very simple devices. The Connection Machine of Thinking Machines Corp. - a 65,536 processors SIMD computer - is then used to run the network.

The primary motivation for the research presented in this thesis has been the design and implementation of rule-based expert system, in a parallel environment, for the sequent Balance 8000. Many of the research presented above have concentrated on expert systems written in OPS5, and more specifically on the parallelization of the RETE match algorithm. However, these systems were optimized to run on serial machines, and much effort and skill went into making the system as deterministic as possible (many choices were pre-eliminated to minimize the searching time) which results in making these systems more difficult to parallelize. Furthermore, this area becomes well explored, and any new research is unlikely to produce a significant speed up.
Our approach is concerned with paradigms other than those available under Prolog, Lisp or OPS5. We believe that an expert system that is written with parallel processing in mind would yield a higher degree of useful parallelism and would also be easier and faster to write than those sequential systems.

As mentioned before, a rule-based system may operate in either forward-chained (data driven) or backward-chained (goal-driven) mode. In the next chapter, we describe them in the forward-chaining mode, while in Chapter 6 we focus our attention on backward-chaining systems.

4.10 Other Research Topics in Expert Systems

(i) Knowledge Acquisition:

This is the process of collecting potential problem-solving expertise from some reliable knowledge source and then transferring and transforming this knowledge to a program. Learning programs associated with expert systems differ considerably in the extent to which the program acquires knowledge by being told, by modifying or manipulating what it already knows, by induction from some set of examples, or by discovering new concepts. Many aspects of machine learning from relevant to expert systems are still topics for research, e.g. learning from incomplete and uncertain information, learning from errorful descriptions and learning general rules with exceptions.

(ii) Knowledge Representation:

It is concerned with the way in which information might be stored in the human brain, and the (possibly analogous) way of storing conveniently large bodies of
knowledge in data structures for the purposes of symbolic (non-numeric) computation.

(iii) Knowledge Application:

Expert system design involves paying close attention to the details of how knowledge is accessed and applied during the search for a solution. Knowledge about knowledge e.g. when and how to use it, is usually termed Meta-knowledge. Different strategies for bringing domain-specific knowledge to bear will generally have marked effects upon the performance characteristics of programs. Most knowledge representation formalisms can be employed under a variety of control regimes, and expert systems researchers are continuing to experiment in this area.

(iv) Generating Explanations:

It is concerned with how to help a user to understand the structure and function of some complex piece of software. The contribution of expert systems researchers to date has been to place a high priority upon the accountability of consultation programs, and to show how explanations of program behaviour can be related to the chains or reasoning employed by rule-based systems. Ongoing contributions include attempts to separate out the different kinds of structural and strategic knowledge implicit in expert performance, and attempt to make explicit and accessible the design decisions associated with the specification of consultation programs for the purposes of generating better explanations and automatic programming. This issue is sometimes called "Transparency".
(v) Educational applications:

This is the study of the possibility of using expert system as an educational tool. Research has shown that it is by no means straightforward to adapt a running expert system as an intelligent tutor.

The reasons for this include our ignorance of: what makes for a productive tutorial dialogue; what constitutes a good explanation for pedagogical purpose; and how to mix conventional telling and testing with the more exploratory modes of learning by experiment and discovery.
CHAPTER V

The Parallel Forward Chaining Models: Design and Implementation
5.1 Introduction

Rule-based systems use three reasoning mechanisms in their search for a solution. These are:

1. Forward chaining.
2. Backward chaining.
3. Mixture of both Forward and Backward chaining.

In this chapter we limit the discussion to the forward chaining mechanism.

As mentioned before, a Rule-based system consists of three main parts:

1. The Rule Base (RB)
2. The Data Base (DB)
3. The Inference Engine (IE).

The Rule Base maintains the set of rules used by the system. The Data Base contains the known facts about the problem, as well as the run-time derived data or facts. Finally, the inference engine controls the whole operation of the rule-based system. In this chapter we describe the design and implementation of eight parallel forward chaining models, together with a detailed study of their efficiency.

The eight parallel models are:

1. **PFC-S1**: Parallel forward chaining (technique 1) with static scheduling, and with the database (DB) retained in the shared memory.

2. **PFC-SM1**: Parallel forward chaining (technique 1), with static scheduling, and with local copies of DB kept in each processor's local memory, and a main copy stored in the shared memory.

3. **PFC-D1**: Parallel forward chaining (technique 1), with dynamic scheduling, and with DB retained in the shared memory.
4. **PFC-DM1**: Parallel forward chaining (technique 1), with dynamic scheduling, and with local copies of DB kept in each processor’s local memory, and a main copy stored in the shared memory.

5. **PFC-S2**: Parallel forward chaining (technique 2), with static scheduling, and with DB retained in the shared memory.

6. **PFC-SM2**: Parallel forward chaining (technique 2), with static scheduling, and with local copies of DB kept in each processor’s local memory, and a main copy stored in the shared memory.

7. **PFC-D2**: Parallel forward chaining (technique 2), with dynamic scheduling, and with DB retained in the shared memory.

8. **PFC-DM2**: Parallel forward chaining (technique 2), with dynamic scheduling, and with local copies of DB kept in each processor’s local memory, and a main copy stored in the shared memory.

### 5.2 The Forward Chaining Process

Forward chaining in its simplest form is an interactive program that performs a loop of substitution. It steps through the rule base (RB) until it finds a rule in which the premises (conditions) match the facts or situation. The rule will then be used or ‘fired’ and a new fact inserted. Once the rule has been used, it will not be used again in the same search; however, the fact which is concluded as the result of the rule’s firing will be added to the data base (DB). This cycle of finding an applicable rule, firing it and adding the conclusion to DB will be repeated until no more matched rules can be found.

The forward chaining algorithm cycles through three basic steps:
1. **Match and find:** In this first step, the left-hand sides (conditions or premises) of all productions (rules) are matched against the contents of the database to determine if the rule is applicable. As a result we obtain a set $S$ (conflict set) of applicable rules which consists of all the rules whose conditions are satisfied by the state of the database.

2. **Select:** Choose one (or more) rules from the set $S$ for execution, according to some predefined criterion of the control scheme.

3. **Act:** Perform and execute the actions in the right-hand side (conclusions) of the selected rules. This action can alter the database by adding assertions to it, or deleting assertions from it.

   Amongst these three steps, the matching and location step is the most computationally intensive. This is because every condition of every rule has to be matched with every element of the database, and this process is repeated again and again after every firing step. Usually 90% of the time in executing production systems on a uniprocessor machine is spent in matching [43]. An important point to note is that most systems use control strategies that prevent them from doing this matching in every cycle. In technique 2 the models are provided with such control strategies.

### 5.3 Levels Of Parallelism in Forward Chaining Technique

A forward chaining program consists of three principal modules:

1. **Module 1: (collecting-facts module)**

   Which does the user interface, asks the user questions to collect the facts about the subject in hand. This module is arranged separated from the others in
the forward chaining technique used here, and it is only activated once at the beginning of the consultation, i.e. the first step in the forward chaining program.

2. Module 2: *(find applicable rules module)*

Which does the matching operations in order to determine the set of applicable rules in each cycle. It completes the following steps:

FOR each rule to be tested, DO:

- Match the conditions of the rule against the database elements.
- If the conditions are satisfied, add the rule to the set of applicable rules (S).

3. Module 3: *(select-fire module)*

- SELECT the rules (or rule) to fire and fire these rules (this rule) i.e. execute their actions. This action updates the database.
- DEFINE the new set of rules to be tested in the next cycle.

In this section, the parallelism of forward chaining is studied and divided into five levels: rules level, condition level, action level, firing-rules level and finally match-test level.

We now discuss the various levels in the execution of forward chaining at which the parallelism can be achieved:

(1.) *Rules Level:*

This is to partition the rules (of the rule base) amongst the processors to be processed in parallel (These are tested to decide their applicability). On a uniprocessor machine all the rules are to be processed by the unique processor (Figure 5.1a). This is a very time-consuming operation. The idea of this level of
parallelism is to distribute the rules among the processors of the multiprocessor machine in a way that each processor processes its assigned share of the rules. By "process the rules" we mean the following actions: match the conditions of the rule against the database elements, determine the applicability of the rule and add the rule to the set $S$ of applicable rules once the rule is applicable. Figure 5.1b illustrates the rules level parallelism.

Note that the rule partitioning is done using the data-partitioning method (homogenous multitasking) on the Balance with static scheduling (Figure 5.2a) and dynamic scheduling (Figure 5.2b). This will be explained further in the next section.

(2.) Conditions Level:

All the conditions within a rule can be processed at the same time to see if they are satisfied by the database elements i.e. match the data or not. This is illustrated in Example 1 and Figure 5.3.

Example 1: $r_k$:

$$\text{IF } C_1 \& C_2 \& \ldots \& C_j \ldots \& C_p$$

$$\text{THEN } A_1, A_2, \ldots A_q$$

$C_{i,i=1,p}$ are the conditions of the rule $r_k$.

Note that we can always distribute the conditions among the processors in a way that 2 or more conditions are to be processed by one processor. The number of processors allocated for the job will be divided by 2 or more depending on the number of conditions/processors. This parallelism level is a fine grain parallelism.
(3.) Actions Level:

After an applicable rule is selected to fire, parallelism in its RHS (right-hand side) can be explored to speed up the execution. This parallelism level is a fine grain parallelism, and unless a lot of attention is made in the distribution of the action among the processors or the action itself needs a reasonable execution time then the implementation of this level of parallelism will not yield great gains in speedup. On the contrary, if the time taken by the system to set up the processors,
are the rules

\[ p_{i+1} = r_m \]

Figure 5.2a: Static scheduling (data partitioning)

Each processor does the following cycle:
1. checks the shared QUEUE and verify that there is still work (task) to be done.
2. removes one or more tasks from the queue.
3. execute those tasks and return to step (1) for more work.

Figure 5.2b: Dynamic scheduling (data partitioning)

Figure 5.3: Conditions level parallelism
assign the actions $A_i$'s to them and then to synchronize the work is greater than that of processing those $A_i$ sequentially on a uniprocessor system, there will be a degradation of the system and no apparent gain.

(4.) Firing-rule Level (multiple firing):

Several rules can be fired simultaneously to shorten the sequence of firings. If the set of selected rules contains more than one rule, they can be fired in parallel (simultaneously). A similar remark for action level is also applicable.

(5.) Match-test Level:

Suppose we have a rule $r_p$. $r_p$: IF $C_1$&...&$C_t$ THEN $A_1$, ..., $A_k$.

To test the applicability of $r_p$, the $C_i$'s are matched against the database elements. This process is repeated for each condition $C_i$ and against all the database elements. (Suppose we do not use techniques like the RETE or other matching algorithms). This operation is repeated for each rule. Matching the conditions $C_i$'s with database elements can be done in parallel to speed up this process (match process).

5.4 The Parallel Models: Design and Implementation

In this section we discuss the parallel forward chaining models together with their implementations and multiprocessing. The forward chaining models are for a rule-based expert system. All the models are domain independent. To support the use of these models, a “rulebase compiler” has been built to translate a rule base in text format into the data structure needed by the system.

The production rules are stored (loaded) in the shared memory of the system. They are distributed amongst the processor's local memories, during execution time,
according to a data-partitioning method with both dynamic and static scheduling. (See rules-level parallelism and Chapter 3). The rules which are loaded in a processor do not migrate to other processors during execution.

For the database there are two choices:

1. The first choice is to keep one full copy of the database in each processor's local memory and a main copy in the shared memory. This main copy is updated after each cycle (firing), and the changes are broadcast through the bus to every processor to update its own local copy. This is illustrated in Figure 5.4a.

2. The second choice is to keep one full copy of the database in the shared memory. This copy will be updated after each cycle (firing). It is used (accessed) by all the processors in their matching process as shared data. This shared copy is made as read-only array data i.e. independent data (see Chapter 3). This is illustrated in Figure 5.4b.
Figure 5.4a: Local copies of the database are kept in each processor's local memory and a main copy stored in the shared memory.

Figure 5.4b: Database kept in shared memory and accessed by the processors as shared data.
5.4.1 PARALLEL MODELS USING TECHNIQUE 1

5.4.1.1 The PFC-S1 Model

(Static scheduling with DB retained in the shared memory)

(I) The Data Structures Used

Two main data structures are used; these are the rulebase and the database structures.

(1.) The Rulebase:

The structure used for the rulebase is an array of records, where each record represents a rule. Each rule is then a record structure which has 3 attributes: the number of conditions (premises) in the rule (this is an integer called ‘Nprem’), the list of premises or conditions, and the list of conclusions or actions. The conditions are arranged in an array of records called ‘premise’. The reason behind using the array structure rather than a linear list structure is that in an array, each element can be accessed separately and individually, which makes it more suitable than a linear list for parallel implementation.

The same ‘array of record’ structure is used for the conclusions. An example of a rule structure is the following:

Example 2: \( r_1 : \text{IF } C_1 \text{ and } C_2 \text{ and } C_3 \)

THEN \( A_1, A_2 \).

This rule is represented internally as a record which has:

(i) The rule number is equal to one.

(ii) The list of premises is an array with 3 elements: \( C_1, C_2 \) and \( C_3 \).

(iii) The list of conclusions is an array with 2 elements: \( A_1 \) and \( A_2 \).
This is illustrated in Figure 5.5.

![Figure 5.5: Internal representation of rule 1](image)

(2.) The Database:

The Database is represented as an array of records called "object-record". Each record represents a fact and contains information such as: the object name, the value and some other useful information.

(II) The Model and its Implementation

We assume that:

- The rules are numbered from 1 to n.
- The processors are numbered from 1 to m.
- The rules are kept in ascending order of the number of conditions in each rule. This is done to balance the work amongst the processes at run-time and as a consequence minimises the idle time for each processor.
- The facts needed by the system throughout its search for a solution are collected by the "User-Interface" module at the beginning of the system run-time. This is done for performance measurement reasons.
The model is illustrated in Figure 5.6.

The model consists of three main modules. These are:

1. The Forward-chaining (FC) module (The supervisor): This organizes and controls the overall activity of the forward chaining system. This module supervises the other two modules which work under its control.

2. Finding Applicable Rules (FAR) module: This completes the matching operation which compare the left-hand sides (condition part) of the rules with elements of the database to determine the applicability of rules. The applicable rules will be kept in a set $S$, together with some other useful information.

3. Select-Fire (SF) module: This selects the rules (rule) to fire and fires them, i.e. execute their actions, and update the database accordingly.

Figure 5.6: The main modules of PFC-S1
The FC module defines the set of rules to be processed, and sends these rules to the FAR module for match testing. The FAR module processes the rules and add the successful rules to the set $S$. The set $S$ is then sent to the SF module. Later the SF module selects some rules (one rule) to fire, fire the rules (rule), and update the database accordingly. The FC module will then test the results and accordingly decide to stop the execution and give results of the consultation, or define the new set of rules to be processed by the FAR module. In the next section each module is discussed in detail together with the system implementation.

(III) Algorithm Structure and Implementation

The PFC-S1 model concept is depicted in Figure 5.7. The FC is the main module which controls and supervises the overall activity of the system. The processes communicate with FAR to receive a job to execute. Each job consists of one or more rules to investigate. The processes receive the jobs from FAR (through the shared memory), process these jobs, and then returns the results of their investigation to FAR, which consequently updates the set of applicable rules. The processes also communicate with the shared database copy, during the matching phase.

The process cycle is to:-

- Receive a task from FC.
- Process the task. This involves matching the conditions against the database elements.
- IF the result of the match-test is positive THEN
  - Call a lock

143
- Report the result to FC, and subsequently to FAR
- Unlock the lock
  - Ask FC for more work.

The system cycles through three principal phases:

**Phase 1a:**

Define the set $R_i$ of rules to be sequenced for the match test. This is the set of rules that has not been fired in the previous cycles (0 to $i - 1$) of the same session.

Let $r_j$ be a rule:

$$r_j : \text{IF } C_1 & C_2 & C_3 \ldots & C_p$$

**THEN** $A_1, A_2, A_3, \ldots A_q$

Suppose we are in the $i^{th}$ cycle of the consultation (inferencing process).

Let $F_i$ be the set of the rules that were fired, i.e. their actions were used and performed, during cycles 0 to $(i - 1)$ of the same consultation.
\( F_i = \{ r_j / \text{its actions } A_{k=1,i} \text{ have been performed during the} \) \\
\( \text{cycle } C_{\ell}/0 \leq \ell < i, \text{ in the same consultation} \} \equiv \) \\
\( \{ r_j / r_j \text{ was fired during a cycle } C_{\ell}/0 \leq \ell < i \}. \)

We define \( R_i \) as: \( R_i = \text{Rulebase} - F_i; \)

In other words, \( R_i \) is the set of all unused rules, (not fired) in cycles 0 to \((i - 1)\) of the consultation.

Note that: \( R_0 = \text{Rulebase}; \) and \( F_0 = \{ \}. \)

The set \( F_i \) is presented internally by an array of records. Each record contains the fired-rule number and other information. The array is called 'Fired rules', and it is updated by the SF module at the end of each cycle (add more fired rules).

**Phase 1b:**

So the first phase is to define the set \( R_i \) of rules to be tested and then to send these rules to the FAR module for processing.

**Phase 2:**

Process the rules in \( R_i \) and form the set \( S \) of applicable rules. This operation is the most important one, since it consumes much of the system overall run-time.

The set \( S \) is kept in the shared memory of the system. This will maintain the applicable rules in the current cycle of the consultation. The data structure of \( S \) is shared by all the processes and is used as write-only data by these processes.

The general algorithm for this phase is the following:

1. Define the number of processes to be used to execute the job. This number is either defined by the user at the start of this phase, or is considered by the
system as the default value of \((\text{number of CPUs on-line})/2\). Assume that this number is defined as equal to \(q\), with \(1 \leq q \leq m\).

2. Create \(q\) processes to carry out the job.

3. Assign to each process a task to carry out i.e. partition the rules amongst the \(q\) processes. This partitioning of the rules amongst the processes is done according to the data-partitioning method with static scheduling. (See Figure 5.2a).

4. Each process processes its share and then waits until other processes finish the execution of their tasks.

5. If there is more work to do, then go to step 3. Otherwise release the processes and either store them for another use, or terminate them (kill the processes).

These steps are further explained with more detail in the following subsections.

**Step 1:**

Read in the number of processes to be set up for the work in hand. This variable is entered by the user at the start of the inferencing process. This number should be greater than zero and less than or equal to the maximum number of processes the system can create. Note that the maximum number of processes created by the system is the total number of processors on line minus one.

In the case that no number is defined, the system will default the number of processes to \((\text{number of CPUs on-line})/2\).

**Step 2, 3 and 4:**

Suppose that the number of processes defined in step 1 is \(q\). The system will then set up \(q\) processes for the work. This is done by calling a parallel programming
library PPL routine called "m-set-procs". This routine declares the number of processes to execute the work in parallel on subsequent calls to a (PPL) routine called "m-fork". A call to m-set-procs (q), will allocate q processes for the execution of the work in parallel. Following this step, a call to the routine "m-fork" will create q processes and assign the "inferencing subprogram" to the child processes, which then cooperates in executing this subprogram in parallel.

The system creates q processes, one called the "Parent Process" (the first process created) and (q - 1) "child processes" which are full copies of the parent process. Each process is given a private integer variable called "m-myid", which identifies it within the set of processes being created. The parent process's identification number (ID) is always zero. The first child process's ID is 1, the second is 2, and so on. A process's ID can be found out by calling the PPL routine "m-get-myid", which return the ID value of the process. The "m-fork" routine assigns the work to the q processes, which then cooperate in executing the work in parallel. The partitioning of the rules amongst the processes is done according to the data-partitioning method with static scheduling of the processes. The static scheduling algorithm simply divides the tasks evenly among the processes. The tasks are scheduled by the processes at run-time, but they are divided in some predetermined way.

Each task consists of:

1. A number of rules to be processed by a process. This number is considered as equal to 1 in the discussion here.

2. Processing the rule and determine the applicability or not of this rule.
3. Report the result to FAR and update $S$ accordingly.

We define the "task-size" as the number of rules in the task. Consequently,

\[ \text{task-size} = 1. \]

The static scheduling algorithm for a process is:

- Figure out which tasks, I will do.
- Do all my tasks.
- Wait until all other processes finish their tasks.

In our model, this algorithm is applied in the following way (see Figure 5.8):

1. Call the (PPL) routine "m-get-numprocs" to determine how many processes were created to perform the job. (This number is $q$).

**Cycle 0**

2. FOR each process DO

   a) Call the PPL routine "m-get-myid" to find out the process's ID. (Call this number $N$).

   b) Define the task to do. In other words, figure out which rule to process. This is determined in the following way:

   \[ I = \text{The rule number} = N + \lambda q, \text{ where } 0 \leq \lambda \text{ and } N + \lambda q \leq n. \]

   at cycle 0, $\lambda = 0$, at cycle 1, $\lambda = 1$; and so on.

   c) IF $I > n$ THEN no work is done and the process becomes idle.

   **ELSE**

   - Assign the $I^{th}$ rule to the process (with ID equal to $N$).
   - Process the $I^{th}$ rule.
   - IF the rule is applicable THEN
- Lock a lock
- Report the result to FAR and update the set S in shared memory.
- Unlock the lock.
- Wait for the other \((q - 1)\) processes to finish their tasks.

3. All the \(q\) processes have finished their tasks.

4. IF there is more work to be done, i.e. more rules to be processed, or the goal is not yet achieved, THEN
   - Start the next cycle.
   - Repeat steps 2 to 4 for the \((n - q)\) rules left unprocessed.

To clarify more this idea, consider the example shown in Figure 5.8:

Cycle 0 \((\lambda = 0)\):

**Step 1** Assign \(r_1\) to \(P_1\), \(r_2\) to \(P_2\), \ldots, \(r_j\) to \(P_j\), \ldots, \(r_q\) to \(P_q\).
Step 2: \( P_1 \) processes \( r_1 \), \( P_2 \) processes \( r_2 \), \ldots, \( P_j \) processes \( r_j \), \ldots, \( P_g \) processes \( r_g \).

Cycle 1 \((\lambda = 1)\):

Step 1: Assign \( r_1 \) to \( P_1 \), \( r_{2+q} \) to \( P_2 \), \ldots, \( r_{j+q} \) to \( P_j \), \ldots, \( r_{2q} \) to \( P_g \).

Step 2: \( P_1 \) processes \( r_{1+q} \), \( P_2 \) processes \( r_{2+q} \), \ldots, \( P_j \) processes \( r_{j+q} \), \ldots, \( P_g \) processes \( r_{2q} \).

The operation will be repeated for more cycles until the goal is achieved or the rules have been all processed.

Step 5:

After the processes finish the execution of the inferencing subprogram (work), the forward chaining module returns from the m-fork call and the child processes spin, waiting for more work. The FC module, can either kill the child processes with a call to the routine m-kill, suspend them with a call to m-park-procs, or let the child processes spin until they are re-used by another m-fork call. If the child processes are to be re-used, the m-park-procs offers the most efficient use on the Sequent system, because it saves the CPU usage of having the processes spin and it saves the overhead of having to recreate processes on the next m-fork call.

The Process Form

Each process has the following form:

1. Local variables and data:

These are the data and variables local to each processor. They are kept in each processor's local memory and accessed only by the processes which run on this processor. The main local variables for each process are the following:
(i) A variable called “rule”. This serves as a buffer to hold one rule structure only.

(ii) A variable called “rule-I” which holds the number of the rule i.e. the index of the rule in the rulebase array in shared memory.

(iii) A variable called “process-id” which holds the process identification (ID) number.

(iv) A Boolean variable called “match” which is used to hold the result of the match-test for the rule.

In addition to these variables and data, each process contains the essential software (procedures and functions) to perform the match test of the rules and other tests, to send and receive data from shared memory, and finally to synchronize with the other running processes.

2. Global shared variables and data:

These are the variables and data resident in the shared memory of the system and accessed by all the processes. Each process uses the following global-shared data:

(i) The Rulebase

(ii) The Database

(iii) The set $S$ of all applicable rules

(iv) A variable called “index-table” which serves as an index to $S$.

3. Process algorithm:

The algorithm for each process is the following:

a) Call the parallel programming library (PPL) routine “m-get-myid” to determine the process ID. This value will be stored in the local variable “process-id”. (We will call this number $N$).
b) Call the PPL routine "m-get-numprocs" to determine how many processes were created by the "m-fork" call i.e. the number of processes to process the work. (We will call this number $q$).

c) Start by processing the $N^{th}$ rule:

1. Load the $N^{th}$ rule into the process local memory. (rule-I← $N$)

2. Test all the conditions of the rule against the database elements in the shared memory to determine the applicability of the rule. If all the conditions are satisfied then set the local variable "match" to TRUE, else set it to FALSE.

3. IF match = TRUE THEN

   - Create a critical section by setting a lock, this is done by calling the PPL routine "m-lock". This routine is used to lock the section of code which comes between the "m-lock" and "m-unlock" calls. It makes sure that only one process can execute this code at the same time.

   - Increment the value of the shared variable "index-table" by 1;
     (index-table ← index-table +1).

   - Add the rule number (rule-I) to the set $S$ of applicable rules, at the (index-table)$^{th}$ element of $S$, i.e. {$S[index-table] ← rule-I$}.

   - Unlock the lock. This is done by calling the PPL routine "m-unlock".

d) Process every $q^{th}$ rule until all the rules are processed. In other words, process the rules: $N + q, N + 2q, N + 3q, \ldots$, until we reach the end of the rulebase, i.e. no more rules to process. (See Figure 5.8).
Phase 3: (Select-Fire phase)

SELECT a number of rules from the set $S$ of applicable rules and fire them. The algorithm for this phase is:

1. Select $\ell$ rules from the set $S$.
2. IF $\ell = 1$ THEN fire the rule separately.
3. ELSE IF ($1 < \ell \leq m$) THEN
   a) Allocate $\ell$ processors, using the “m-set-numprocs” routine.
   b) Create $\ell$ processes. Each process’s task is to fire one rule, i.e. execute the rule’s actions (conclusion part) and update the database accordingly. So, Process 1 fires the rule $S[1]$, process 2 fires rule $S[2]$ and so on.
   c) Each process executes its own task.
4. ELSE ($\ell > m$)
   a) Create $m$ processes.
   b) For each process do:
      - Call the routine “m-get-myid” to determine the process ID. (Call this number $N$).
      - Call the routine “m-get-numprocs” to determine the number of processes created by the “m-fork” call. (This number is $m$).
      - Start by processing the $N^{th}$ element of $S$, i.e. fire the rule $S[N]$.
   c) Fire every $m^{th}$ rule in $S$ until we reach the end of $S$. Thus, fire the rules: $S[N + m]$, $S[N + 2m]$, and so on until all the rules in $S$ are processed.
In the work presented in this section, the selection strategy used is to fire all the rules in $S$ at the same time. This is possible, because during Phase 2 the FAR module filtered out all those applicable rules which would duplicate the data in the database, if they are fired.

Finally, the set $S$ is initialized to the empty set, and the variable "index-table" is initialized to zero at the beginning of each FAR cycle.

5.4.1.2 Performance Analysis of PFC-S1

As mentioned before, the system used to implement the parallel models is the Sequent Balance 8000. This is a shared-memory (Tightly coupled) multiprocessor system. The maximum number of processors available on the system at PARC (Parallel Algorithm Research Centre) is 10 processors, one of these processors is used by the operating system, and only 9 processors are effectively available to run parallel applications.

Theoretically, a program running on a multiprocessor system with $p$ processors should run $p$ times faster than the same program running on a single processor. In practice, however, the speed-up is often smaller, mainly because of one or more of the following degrading factors:

1. The overheads due to synchronization and scheduling in the parallel implementations.
2. The processes communication overheads.
3. The bus-access and shared-memory contention.

The models are tested with a different number of processors (between 1 and 9), so as to find out the speed-up obtained from each of the parallel implementations.
The experiments are carried out for different rulebases, with different numbers of rules, as to study the effect on speed-up and efficiency of the problem sizes and complexity, i.e. the effect of the number of rules on the performance of the parallel models.

Let $T(p)$ be the time spent by $p$ processors to execute a given algorithm and $T(1)$ be the time-complexity of the same algorithm running on one processor. Then the speed-up ($S_p$) is defined as:

$$ S_p = \text{SPEEDUP} = \frac{T(1)}{T(p)} \leq p $$

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

$$ E = \text{EFFICIENCY} = \frac{S_p}{p} = \frac{T(1)}{p \times T(p)} \leq 1 $$

Note that, in the test carried out here we do not time the initial loading and set up of the knowledge base (rules and facts). Only the process of inferencing is timed.

Tables 5.1 and 5.2 show the experimental results obtained for the parallel model PFC-Sl. Their graphical interpretations are shown in Figures (5.9) and (5.10) respectively.

The experiments were carried out using two sets of rules with different sizes. This was to study the effect of the problem size and complexity on the performance of the parallel model in the execution of expert systems.

It can be observed from the experiments that the speedup and the efficiency are better when the size of the rulebase is larger. On the other hand, the efficiency index for small numbers of processors (1 to 5) indicates an acceptable performance level, however for larger numbers of processors the converse is true.
<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Computing Time (msec)</th>
<th>SPEEDUP</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>94890</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>48620</td>
<td>1.952</td>
<td>0.976</td>
</tr>
<tr>
<td>3</td>
<td>34480</td>
<td>2.752</td>
<td>0.917</td>
</tr>
<tr>
<td>4</td>
<td>27340</td>
<td>3.471</td>
<td>0.868</td>
</tr>
<tr>
<td>5</td>
<td>23030</td>
<td>4.120</td>
<td>0.824</td>
</tr>
<tr>
<td>6</td>
<td>20570</td>
<td>4.591</td>
<td>0.765</td>
</tr>
<tr>
<td>7</td>
<td>19190</td>
<td>4.945</td>
<td>0.706</td>
</tr>
<tr>
<td>8</td>
<td>17720</td>
<td>5.355</td>
<td>0.669</td>
</tr>
<tr>
<td>9</td>
<td>17110</td>
<td>5.546</td>
<td>0.616</td>
</tr>
</tbody>
</table>

Table 5.1: Experimental Results for the Parallel Model PFC-S1. [Rulebase = 350 rules].

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Computing Time (msec)</th>
<th>SPEEDUP</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>119020</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>61110</td>
<td>1.948</td>
<td>0.974</td>
</tr>
<tr>
<td>3</td>
<td>42620</td>
<td>2.793</td>
<td>0.931</td>
</tr>
<tr>
<td>4</td>
<td>33310</td>
<td>3.573</td>
<td>0.893</td>
</tr>
<tr>
<td>5</td>
<td>28140</td>
<td>4.230</td>
<td>0.846</td>
</tr>
<tr>
<td>6</td>
<td>25420</td>
<td>4.682</td>
<td>0.780</td>
</tr>
<tr>
<td>7</td>
<td>23100</td>
<td>5.152</td>
<td>0.736</td>
</tr>
<tr>
<td>8</td>
<td>20790</td>
<td>5.725</td>
<td>0.716</td>
</tr>
<tr>
<td>9</td>
<td>19870</td>
<td>5.990</td>
<td>0.666</td>
</tr>
</tbody>
</table>

Table 5.2: Experimental Results for the Parallel Model PFC-S1. [Rulebase = 450 rules].
Figure 5.9: The speedup curve for the parallel model PFC-S1. 

\[ \text{Rulebase} = 350 \text{ rules} \]

Figure 5.10: The speedup curve for the parallel model PFC-S1. 

\[ \text{Rulebase} = 450 \text{ rules}. \]
5.4.1.3 The PFC-SM1 Model

*Static scheduling with local copies of DB kept in each processor's local memory.*

(I) The Model and its Implementation

The same data structures used for PFC-S1 are again used for PFC-SM1. The model consists of three main modules. These are:

1. The Forward-chaining (FC) module. (The supervisor)
2. The Finding Applicable Rules (FAR) module.
3. The Select-Fire (SF) module.

The PFC-SM1 model concept is depicted in Figure 5.11. The FC is the main module which controls and supervises the overall activity of the system. A local copy of the database is kept in each processor's local memory, and is used by the process in the matching phase of the inferencing mechanism. The processes communicate with FAR to receive a job to execute. Each job consists of one or more rules to investigate. The processes receive the jobs from FAR, process these jobs, and then returns the results of their investigation to FAR, which consequently updates the set $S$ of applicable rules. The set $S$ is sent to the SF module. Later the SF module selects some rules (one rule) to fire, fire(s) the rule(s) and updates the main copy of the database (in shared memory). The changes in the main copy (DB) are then broadcast by FAR to each processor to update its own local copy of the database. This is illustrated in Figure 5.12.

The model cycles through the following steps:

1. Collect the facts needed for the consultation.
2. Create a local copy of the database in each processor's local memory.
3. Define the set $R_i$ of rules to be processed. $R_i =$ rules that have not been fired during the cycles $0$ to $i - 1$.

4. Process the rule in $R_i$ and form the set $S$ of applicable rules.

5. Select rules to fire.

6. Fire the rules and update the main-copy of the database DB accordingly.

7. Broadcast the changes in the main-copy DB to the processors in order to update their own local copies of the database LDB.

8. Test the results of the present cycle and accordingly decide whether to stop the execution and give the results of the consultation or to start another cycle, i.e. go to step 3.

After firing the rules at step 6, all the changes to the Main Copy DB are maintained in an array (update-array), which have the same structure of DB. This "update-array" is then used in step 7, by the FAR module to update the processor's local copies LDBs.

(II) The Process Form

Each process, running on one processor, has the following form:

1. Local variables and data:

   These are the data and variables local to each processor. The main local variables for each processor are the following:

   i) The variables: $rule$, $rule-I$, $process-id$, $match$, used in the model PFC-S1 are again used here for PFC-SM1.

   ii) A variable called $LDatabase$ which is used to store the local copy of the database. $LDatabase$ is an exact copy of the DB structure and is kept in the processor local memory.

   159
Databuc
FAR module
- - - control flow
--- dataflow
LMI: Locomotor Memory
Pi: Processor i
LDatabase: Local Copy of the Database
1. Define rules to process
2. Find the set S
3. Fire and update DB
4. Broadcast the changes in DB to LDBs
(Local copies of the Database)

Figure 5.11: The PFC-SM1 system concept

Figure 5.12: The PFC-SM1 cycle
In addition to these variables and data, each processor contains the essential software (procedures and functions) to process the task in hand.

2. Global shared variables and data:

These are the same used for PFC-S1. They are: the rulebase RB, the main-copy of the database DB, the set $S$ of applicable rules and the variable index-table. In addition to these, there is a new variable called update-array, which has the same structure of DB, and is used to maintain the changes to the DB copy, resulting from firing applicable rules at each cycle. This variable is used by the FC module as a "reference copy" to update the processor's local copies $LDatabase$.

3. Process algorithm:

The algorithm for each process is the following:

a) Call the parallel programming library (PPL) routine "m-get-myid" to determine the process ID. This value will be stored in the local variable process-id. (We will call this number N).

b) Call the PPL routine "m-get-numprocs" to determine how many processes were created by the "m-fork" call. (We will call this number $q$).

c) Start by processing the $N^{th}$ rule:

1. Load the $N^{th}$ rule into the process local memory (rule-$I-N$).

2. Test all the conditions of the rule against the local copy of the database LDB, to determine the applicability of the rule. If all the conditions are satisfied then set the local variable match to TRUE, else set it to FALSE.
3. IF $\text{match} = \text{TRUE}$ THEN

- Create a critical section by setting a lock. Call the PPL routine $m$-lock.
- Increment the value of the shared variable $\text{index-table}$ by 1;
- Add the rule number (rule-I) to the set $S$ of applicable rules, at the $(\text{index-table})^{th}$ element of $S$, i.e. $S[\text{index-table}] \leftarrow \text{rule-I}$.
- Unlock the lock. Call the PPL routine $m$-unlock.

4. Process every $q^{th}$ rule until all the rules are processed. In other words, process the rules: $N+q$, $N+2q$, $N+3q$, …, until there are no more rules to be processed.

5.4.1.4 Performance Analysis of PFC-SM1

Tables 5.3 and 5.4 show the experimental results obtained for the parallel model PFC-SM1. Figures (5.13) and (5.14) show the graphical interpretations of these tables respectively.

The comparison between the use of shared memory and local memories in the PFC-S1 and PFC-SM1 systems, which is presented in the results given in Tables (5.1), (5.2) and Tables (5.3), (5.4) respectively, clearly shows an increase of the speed-up from 5.99 to 8.279 in experiment 2 (5.546 to 8.136 in experiment 1) and a corresponding increase in efficiency of approximately 30% for the two experiments.

This indicates that the use of local memory in the parallel forward chaining technique is a deciding factor in the design of parallel expert systems. Moreover, it can be observed from the tables that the results are slightly better when the size of the rulebase is larger.
<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Computing Time (msec)</th>
<th>SPEEDUP</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93400</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>47500</td>
<td>1.966</td>
<td>0.983</td>
</tr>
<tr>
<td>3</td>
<td>32050</td>
<td>2.914</td>
<td>0.971</td>
</tr>
<tr>
<td>4</td>
<td>24400</td>
<td>3.828</td>
<td>0.957</td>
</tr>
<tr>
<td>5</td>
<td>19220</td>
<td>4.860</td>
<td>0.972</td>
</tr>
<tr>
<td>6</td>
<td>16430</td>
<td>5.685</td>
<td>0.947</td>
</tr>
<tr>
<td>7</td>
<td>14360</td>
<td>6.504</td>
<td>0.929</td>
</tr>
<tr>
<td>8</td>
<td>12420</td>
<td>7.520</td>
<td>0.940</td>
</tr>
<tr>
<td>9</td>
<td>11480</td>
<td>8.136</td>
<td>0.904</td>
</tr>
</tbody>
</table>

Table 5.3: Experimental Results for the Parallel Model PFC-SM1.  
[Rulbase = 350 rules].

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Computing Time (msec)</th>
<th>SPEEDUP</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>117810</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>59470</td>
<td>1.981</td>
<td>0.990</td>
</tr>
<tr>
<td>3</td>
<td>40380</td>
<td>2.918</td>
<td>0.973</td>
</tr>
<tr>
<td>4</td>
<td>30500</td>
<td>3.863</td>
<td>0.966</td>
</tr>
<tr>
<td>5</td>
<td>24400</td>
<td>4.828</td>
<td>0.966</td>
</tr>
<tr>
<td>6</td>
<td>20490</td>
<td>5.750</td>
<td>0.958</td>
</tr>
<tr>
<td>7</td>
<td>18140</td>
<td>6.494</td>
<td>0.928</td>
</tr>
<tr>
<td>8</td>
<td>15630</td>
<td>7.5370</td>
<td>0.942</td>
</tr>
<tr>
<td>9</td>
<td>14230</td>
<td>8.279</td>
<td>0.920</td>
</tr>
</tbody>
</table>

Table 5.4: Experimental Results for the Parallel Model PFC-SM1.  
[Rulbase = 450 rules].
Figure 5.13: The speedup curve for the parallel model PFC-SM1.  
[Rulebase = 350 rules].

Figure 5.14: The speedup curve for the parallel model PFC-SM1.  
[Rulebase = 450 rules].
5.4.1.5 The PFC-D1 Model

(*Dynamic scheduling with DB retained in the shared Memory*)

The same data structures: Rulebase, Database, and set of applicable rules \( S \) are again used for the PFC-D1. The model consists of three main modules:

1. The Forward-Chaining (FC) module.
2. The Finding-Applicable rules (FAR) module.
3. The Select-Fire (SF) module.

The PFC-D1 model concept is depicted in Figure 5.15.

The FC module controls the overall activity of the system. The FC module communicates with the FAR module to start and control the partitioning of the tasks amongst the processes at run-time. The FC defines the number of tasks (BITE) to be allocated to one process after its request for a job. Each process makes a request for work to FAR, by checking the “task-queue”. If this queue is not empty, FAR will remove a BITE of work from the task-queue and send it to the process. The BITE of work is composed of a predetermined number of tasks from the queue.

This variable BITE is defined by the FC module prior to the partitioning, i.e. at the beginning of cycle 0 of the consultation. The FAR determines the size of the tasks in the queue at the beginning of the first partitioning cycle. The size of a task (task-size) is the number of rules the task has to investigate. To clarify this point consider Figure 5.16:

In this figure the task-queue consists of \( n \) tasks (task 1, … task \( n \)). The size of task 1 is equal to 2, since the number of rules task 1 has to investigate is equal
to 2. In Figure 5.16(a) the BITE of work is defined as consisting of only one task, whereas in Figure 5.16(b) the BITE consists of two tasks to execute.

Each process communicates with the FAR module to receive a BITE of work to process. If the queue is not empty, the FAR will remove a BITE of work from the queue and assign it to the process. If the queue is empty, the process has to wait for some tasks to appear in the queue. If there are no more tasks to process, the FC module will either park the processes for a future use, or terminate them.

The processes receive work from FAR, execute it, and send the results to FAR. The FAR will then update the set of applicable rules $S$ and report these results to the FC module. The FC analyses the result and decide on whether to start another cycle or halt the consultation and give results. The FAR module communicate with the SF module to send the set of applicable rules $S$ at the end of each cycle. The SF will then select some rules and fire them. This action (firing rules) will change the content of DB: add or remove some elements.

The set $R_i$ of rules is treated as a task-queue. Each process removes one or more rules (Bite of work) from the queue, investigates those rules, send the results to FAR, and returns for more work.

This method is called “hungry puppies” because the processes “nibble” away at the work until it is all done. Finally, dynamic scheduling creates more communication overheads than static scheduling because all the processes must access a single shared task queue, but the computing load can be very evenly distributed because no process is idle while there is still work to be done.
Algorithm Structure and Implementation

The PFC-D1 model cycles through the following steps:

**Step 1** Collect the facts used by the system throughout its search for a solution (answer) to the problem (consultation) in hand. These facts are kept in DB in the shared memory of the system.

**Step 2** Define the number of processes to be used to execute the job. This number is either defined by the user, or is considered by the system as the default value of (CPUs on-line)/2. Assume that this number is defined as equal to $q$, with $1 \leq q \leq m$.

**Step 3** Define the set $R_i$ of rules to be sequenced for the match test. This is the set of rules that has not been fired in the previous cycles ($0$ to $i - 1$) of the same session.

$$R_i = \text{Rulebase} - F_i;$$

where $F_i = \{r_j/r_j \text{ was fired during a cycle } C_{\ell}/0 \leq \ell < i\}$

*At cycle 0:*

$$R_0 = \text{Rulebase};$$

$$F_0 = \{\};$$

**Step 4** Define the variable $BITE$. This is the number of tasks to be assigned to one process at one request (see Figure 5.17). In our discussion here, we will consider this number as equal to one:

$$BITE = \text{one task}.$$

**Step 5** Set up the $q$ processors for work. This is done by calling the PPL routine "m-set-nprocs", i.e. call m-set-nprocs ($q$).
Figure 5.15: The FFC-DI model concept

Figure 5.16: The task queue
Step 6 Create $q$ processes to carry out the job. This is done by calling the PPL routine `m-fork`, which creates $q$ processes, the first is the parent process, and the remainder are the "child processes".

Definition 1:

The set $R_i$ of rules is treated as a task queue. Each task consists of a number of rules to investigate. This number is defined by the FAR module at this stage of the consultation. From here on, the terms "Bite" and "task" will be used interchangeably to represent: a certain number $k$ of rules to investigate ($k > 0$).

Definition 2:

Consider a task $T$ which consists of investigating $k$ rules:

$$T \equiv \text{investigate (rule}_1, \text{ rule}_2, \ldots, \text{rule}_k).$$
We define the two variables "Base" and "Top" as:

\[
\text{Base} (T) = 1 \\
\text{Top} (T) = k
\]

In other words, the Base of the task \( T \) is equal to the index of the first rule in \( T, \text{rule}_1 \). The Top of the task \( T \) is equal to the index of the last rule in \( T, \text{rule}_k \).

**Definition 3:**

Processing the task \( T \) is defined as:

\[
\text{Process} (T) \equiv \text{FOR } I = \text{Base to Top } \text{ DO} \\
\quad \text{Process} (\text{rule}_I).
\]

In our discussion here we consider: Bite = one task = 1.

**Step 7** FOR each process DO

a) Call the PPL routine "m-get-numprocs" to determine how many processes were created to perform the job. (This number is \( q \)).

b) Call the PPL routine "m-next" which is used to increment a shared global counter. The first call to "m-next" returns the value 1, the second returns 2, and so on. Assume that the returned value is equal to \( A \).

c) The FAR defines the task \( T, \), removes it from the task queue, and assign it to the process. FAR defines the base and the top of the task \( T, \) in the following way:

\[
\text{Base} = \text{Bite} \ast m-\text{next} \\
\text{Top} = \text{Base} + \text{Bite} - 1
\]

(\( I \))

Note that Bite is considered as equal to 1 in our discussion here.

Thus, by replacing Bite with 1 and m-next with \( \lambda \) we will obtain:

\[
\text{Base} = 1 \ast \lambda = \lambda \\
\text{Top} = \lambda + 1 - 1 = \lambda
\]

(\( II \))
Thus,
\[
\text{Process } (T_\lambda) \equiv \text{ FOR } I = \lambda \text{ TO } \lambda \text{ DO } \equiv \text{ Process } (\text{rule}_\lambda)
\]

\[
\text{Process } (\text{rule}_I).
\]

\[
\text{Process } (T_\lambda) \equiv \text{ Process } (\text{rule}_\lambda)
\]

(III)

In our discussion here we assume that the number of rules which form a task \(T\) is equal to one, i.e. task-size = 1.

d) Process my task \(T_\lambda\). Send the result of the investigation to FAR which updates the set \(S\) of applicable rules. The algorithm for this step is:

- Investigate rule_\lambda.
- IF the rule is applicable THEN
  - Lock a lock
  - Report the result to FAR and update the set \(S\) of applicable rules.
  - Unlock the lock.

Example 3:

Suppose that we want to use 3 processes \(P_1\), \(P_2\), and \(P_3\) to carry out the job in parallel.

Assume that \(P_1\), \(P_2\), and \(P_3\) call m-next in the following order: \(P_1\), then \(P_2\), and then \(P_3\).

Assume that:

- the call to m-next by \(P_1\) returns the value \(\lambda\).

Thus, the call to m-next by \(P_2\) will return the value \(\lambda + 1\)

and, the call to m-next by \(P_3\) will return the value \(\lambda + 2\).

The tasks \(T_\lambda, T_{\lambda+1}, T_{\lambda+2}\) to be processed by \(P_1\), \(P_2\) and \(P_3\) respectively are defined as follow:
Task $T_{\lambda}$:

$$Base = \text{Bite } \ast (m\text{-next}) = 1 \ast \lambda = \lambda$$

$$Top = Base + \text{Bite } - 1 = \lambda + 1 - 1 = \lambda$$

Thus,

$$T_{\lambda} = \text{Investigate (rule}_{\lambda}\text{)}$$

Process $(T_{\lambda}) \equiv \text{Process (rule}_{\lambda}\text{)}$.

Task $T_{\lambda+1}$:

$$Base = \lambda + 1$$

$$Top = \lambda + 2 - 1 = \lambda + 1$$

Thus,

$$T_{\lambda+1} = \text{Investigate (rule}_{\lambda+1}\text{)}$$

Process $(T_{\lambda+1}) \equiv \text{Process (rule}_{\lambda+1}\text{)}$.

In the same way we conclude:

$$T_{\lambda+2} = \text{Investigate (rule}_{\lambda+2}\text{)}$$

Process $(T_{\lambda+2}) \equiv \text{Process (rule}_{\lambda+2}\text{)}$.

The FAR module defines (at each cycle) for each process a task to perform, removes the task from the task-queue, and then assigns the task to the process. Example 1 is depicted in Figure 5.18. In this figure FAR assigns the tasks $T_{\lambda}$, $T_{\lambda+1}$, and $T_{\lambda+2}$ to the processes $P_1$, $P_2$ and $P_3$ respectively. Each process will then process its task and returns to FAR for more work.

Step 8 The FAR reports the results to FC.

Step 9 The SF module fires the selected applicable rules. This action modifies the content of DB.

Step 10 FC takes control, checks the results and acts as follows:

IF no more applicable rules were found ($S = \phi$) THEN GOTO Step 11.
ELSE

IF there is more work to do i.e. more rules to investigate THEN

- Start a new cycle \((i + 1)\).
- Define the set \(R_{i+1}\) of rules to investigate.
- GOTO Step 5.

Step 11 The FC can either terminate the processes with a call to the m-kill routine, suspend them with a call to m-park-procs, or let them spin until they are re-used by another m-fork call.

Step 12 The FC gives the results of the consultation.

(II) The Process Form

Each process has the following form:
1. Local variables and data:

These are the data and variables local to each processor. The main local variables for each processor are the following:

i) The variables rule, rule-I, process-id, match used in the models PFC-S1 and PFC-SM1 are the same used for PFC-D1.

ii) The two variables Base and Top, which define the task to be processed by the process.

In addition to these variables and data, each process contains the essential software to perform the match test of the rules, and other tests, to send and receive data from shared memory, and finally to synchronize with the other running processes.

2. Global shared variable and data:

Each process uses the following global-shared data:

i) The Rulebase

ii) The Database

iii) The set S of all applicable rules

iv) A variable called index-table which serves as an index to S.

v) The variable Bite which defines the number of tasks to be assigned to one process at each job-request cycle.

3. Process algorithm:

The algorithm for each process is the following:

Cycle i:

1. Call the PPL routine "m-get-numprocs" to determine how many processes were created by the FC module. (We will call this number q).
2. Call the PPL routine "m-next". This call will increment a shared global counter \( \text{Counter} \leftarrow \text{Counter} + 1 \) and return a value \( \lambda \).

3. The FAR module defines the task \( T_\lambda \) to be processed by the process as follows:

\[
T_\lambda / \text{Base} (T_\lambda) = \text{Bite} \times \text{m} - \text{next} = 1 \times \lambda = \lambda
\]

\[
\text{Top} (T_\lambda) = \text{Base} + \text{Bite} - 1 = \lambda + 1 - 1 = \lambda
\]

4. The FAR checks the task-queue to find out about the availability of the task \( T_\lambda \). This is done as follows:

\[
\text{IF} \ \lambda > \text{Number-of-rules} (R_i) \ \text{THEN}
\]

- Task is not available
- Wait until all the other \((q - 1)\) processes finish their tasks. GOTO (7)

\[
\text{ELSE}
\]

- Remove the task \( T_\lambda \) from the task-queue, i.e. remove \( \text{rule}_\lambda \) from \( R_i \).
- Assign \( T_\lambda \) to the process.

5. Process the task \( T_\lambda \), i.e. process the rule \( \text{rule}_\lambda \):

- Load \( \text{rule}_\lambda \) into the process local memory.
- Test all the conditions of the rule against DB elements in shared memory, as to determine the applicability of the rule.

\[
\text{IF conditions (rule}_\lambda \text{) are satisfied THEN}
\]

\[
\text{match} = \text{TRUE}
\]

\[
\text{ELSE}
\]
match = FALSE.

- IF match = TRUE THEN
  - Lock a lock.
  - Increment the shared variable "index-table":
    \[ \text{index-table} \leftarrow \text{index-table} + 1. \]
  - Add rule_\lambda to the set of applicable rules S:
    \[ S[\text{index-table}] \leftarrow \text{rule}_\lambda \]
  - Unlock the lock.

6. Repeat the steps (1) through (5) until all the work is finished, i.e. until all the tasks in the task-queue are processed.

7. No more tasks to process (task-queue is empty). All processes finish their tasks.

5.4.1.6 Performance Analysis of PFC-D1

The experimental results obtained for the parallel model PFC-D1 are shown in Tables 5.5 and 5.6. The graphical interpretations of these tables are shown in Figures (5.19) and (5.20) respectively. Again, the experiments were carried out using two sets of rules with different sizes.

As for PFC-S1, it can be concluded from the two tables that the results (speed-up and efficiency) have improved for an increasing number of rules.

By comparing the Tables 5.1 and 5.2 with Tables 5.5 and 5.6 respectively, which represent the timing results for the static and dynamic techniques using the same number of rules, it can be observed that the speed up has improved (with a number of processors greater than 2), resulting in an efficiency increase of 3\% when the rules are allocated dynamically.
<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Computing Time (msec)</th>
<th>SPEEDUP</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93640</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>48580</td>
<td>1.928</td>
<td>0.964</td>
</tr>
<tr>
<td>3</td>
<td>33560</td>
<td>2.790</td>
<td>0.930</td>
</tr>
<tr>
<td>4</td>
<td>26620</td>
<td>3.518</td>
<td>0.879</td>
</tr>
<tr>
<td>5</td>
<td>22640</td>
<td>4.136</td>
<td>0.827</td>
</tr>
<tr>
<td>6</td>
<td>20140</td>
<td>4.649</td>
<td>0.775</td>
</tr>
<tr>
<td>7</td>
<td>18420</td>
<td>5.084</td>
<td>0.726</td>
</tr>
<tr>
<td>8</td>
<td>17180</td>
<td>5.451</td>
<td>0.681</td>
</tr>
<tr>
<td>9</td>
<td>16210</td>
<td>5.777</td>
<td>0.642</td>
</tr>
</tbody>
</table>

Table 5.5: Experimental Results for the Parallel Model PFC-D1. (Rulebase = 350 rules).

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Computing Time (msec)</th>
<th>SPEEDUP</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>118280</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>60700</td>
<td>1.949</td>
<td>0.974</td>
</tr>
<tr>
<td>3</td>
<td>41490</td>
<td>2.851</td>
<td>0.950</td>
</tr>
<tr>
<td>4</td>
<td>32310</td>
<td>3.661</td>
<td>0.915</td>
</tr>
<tr>
<td>5</td>
<td>26990</td>
<td>4.382</td>
<td>0.876</td>
</tr>
<tr>
<td>6</td>
<td>24720</td>
<td>4.785</td>
<td>0.797</td>
</tr>
<tr>
<td>7</td>
<td>22060</td>
<td>5.362</td>
<td>0.766</td>
</tr>
<tr>
<td>8</td>
<td>20410</td>
<td>5.795</td>
<td>0.724</td>
</tr>
<tr>
<td>9</td>
<td>18980</td>
<td>6.232</td>
<td>0.692</td>
</tr>
</tbody>
</table>

Table 5.6: Experimental Results for the Parallel Model PFC-D1. (Rulebase = 450 rules).
Figure 5.19: The speedup curve for the parallel model PFC-D1. 
\(\text{Rulebase} = 350 \text{ rules}\).

Figure 5.20: The speedup curve for the parallel model PFC-D1. 
\(\text{Rulebase} = 450 \text{ rules}\).
5.4.1.7 The PFC-DM1 Model

(Dynamic scheduling with local copies of the Database kept in each processor's local memory).

(I) The Model and its Implementation

PFC-DM1 uses the same data structure used in PFC-D1. A main copy of the database called “Reference Copy” (RCDB) is resident in the shared memory of the system. This reference copy is used by the processes to update their local copies of the database (LDB) at the end of each Match-Fire cycle. The Match-Fire cycle is illustrated in Figure 5.21.

The PFC-DM1 model concept is depicted in Figure 5.22.

A process \( P_i \) communicates with the FAR module to:

a) Initiate a request for a job.

b) Receive a job to process.

c) Report the result of the investigation (Match-test).

The FAR module communicates with the shared memory to:

a) Check the Task Queue for job availability.

b) Read the changes in the Reference Copy RCDB and broadcast these changes to the processors to update their local copies (LDB)s.

c) Add the new applicable rules to S.

The FC module controls the overall activity of the system. At the end of each Match-Fire cycle, FC studies the new situation, i.e. see whether the goal is achieved or not, and if not check if there are more rules to process. Depending on the result of this study, the FC will either start a new cycle or stop the execution and give the results of the consultation.
Start a new cycle

\[ i \leftarrow i + 1 \]

1. Define the set \( R_i \) of rules
2. Set the processors for work
3. The processors process the tasks in \( R_i \) in a dynamic scheduling way
4. Add the applicable rules to the set \( S \)
5. Select rules to fire and Fire them
6. Is the consultation over?
7. Give the consultation results

Figure 5.21: The Match–Fire cycle

---

Figure 5.22: The PFC–DM1 model concept

---

P1

---

control flow
data flow

P1: Processor i

LDB: Local Copy of the Database

LM_i: Local Memory for processor i

S: Shared Memory

FC

---

180
Finally, the SF module communicates with the shared memory to:

a) Receive the set $S$ of applicable rules, at the end of each cycle.

b) Update the content of the Reference Copy RCDB after firing the selected rules.

(II) Algorithm Structure and Implementation

The PFC-DM1 model cycles through the following steps:

Step 1 Collect the facts needed by the system for the consultation. These facts are kept in the Reference Copy RCDB in the shared memory of the system.

Step 2 Define the number of processors to be used for the consultation. (Call this number $q/1 \leq q \leq m$).

Step 3 Create a local copy LDB in each processor local memory.

Step 4 Define the set $R_i$ of rules to investigate. This set of rules is treated as a task-queue, where each task consists of processing one or more rule(s) from the set $R_i$.

Step 5 Define the variable “Bite”. This is the number of tasks to be assigned to one processor at one job-request. In our discussion here, Bite is considered to be equal to one (Bite = one task).

Step 6 Set up the $q$ processors for work.

Step 7 Create $q$ processes to carry out the job.

Step 8 Determine the size of the tasks, i.e. define the task-size (nb. of rules per task). In our discussion here we define the task-size to be equal to one.

(task-size = one rule).

Step 8a Update the local copies LDBs.
IF cycle <> 0 THEN {not the first cycle}.

- Consult the Reference Copy (RCDB) and update the local copies of
  the database LDBs accordingly.

**Step 9** Define and process the tasks.

FOR each process $P_i$ DO:

1. Call "m-get-numprocs" to determine the number of processes created
   by the system for the consultation. (This number is $q$).

2. Call the routine "m-next". This call will increment a shared global
   counter and return a value $\lambda$ ($\lambda > 0$).

3. The FAR defines the task $T_\lambda$, removes it from the task-queue and
   assign it to $P_i$.

   $$T_\lambda \text{ is} / \quad \text{Base}(T_\lambda) = \text{Bite} \times m\text{-next}$$

   $$\text{Top}(T_\lambda) = \text{Base}(T_\lambda) + \text{Bite} - 1.$$  

   Thus, $\text{Base}(T_\lambda) = \lambda$ and $\text{Top} = \lambda + 1 - 1 = \lambda$

   and consequently, $T_\lambda = \text{Process (rule}_\lambda$)

4. $P_i$ processes the task $T_\lambda$, and sends the result of the investigation to
   FAR which updates the set $S$ of applicable rules. The algorithm for this step is:

   a) Investigate $\text{rule}_\lambda$: Match the conditions of the rule against the
      elements of LDB.

   b) IF the rule is applicable THEN

      - Lock a lock
- Report the result to FAR, which consequently adds the rule to S.
- Unlock the lock.

5. Ask for more work. If there is any, go to Step 9.1. This will be repeated until all the tasks in the task-queue are performed.

**Step 10** FAR reports the result of processing the tasks to FC.

**Step 11** SF selects and fires one or more rule(s). This action modifies the content of RCDB.

**Step 12** FC takes control, study the new situation, and acts as follows:

IF no more applicable rules found (S = φ) THEN GOTO Step 13
ELSE
- Start a new cycle (i + 1)
- Define the set R_{i+1}
- GOTO step 6.

**Step 13** FC can either terminate the processes with a call to “m-kill-procs” routine, suspend them with a call to “m-park-procs”, or let them spin until they are re-used by another m-fork call.

**Step 14** FC gives the results of the consultation.

(III) The Process Form

Each process has 3 main constituents:

1. Local variables and data:

These are the data and variables local to each processor and only used by the process running on this processor. The main local variables are the following:
i) The variables: *rule*, *rule-I*, *process-id*, *match* which are used in the models discussed before.

ii) The variables *Base* and *Top* used in PFC-D1.

iii) An array called 'LDB' which is used to store the local copy of the database. LDB has the same structure of DB.

In addition to these variables and data, each process contains the essential software to perform the match test of the rules and other tests, to send and receive data from FAR, and finally to synchronize with the other running processes.

2. Global shared variable and data:

These are the variables and data used by all the processes and resident in the shared memory of the system. They are:

i) The Rulebase

ii) The Reference Copy of the Database (RCDB)

iii) The Task-Queue

iv) The set S of applicable rules

v) The variable *Bite*

3. Process algorithm:

The process algorithm in the PFC-DM1 is the same algorithm described in the PFC-D1 model with only one exception. The match-test for the conditions of a rule is carried out against the elements of the local copy of the database LDB instead of the elements of DB in the shared memory.

5.4.1.8 Performance Analysis of PFC-DM1

Tables 5.7 and 5.8 show the experimental results obtained for the parallel model
PFC-DM1. The graphical interpretations of these tables are shown in Figures (5.23) and (5.24) respectively.

Again, the comparison between the use of shared memory and local memories in the PFC-D1 and PFC-DM1 systems, which is presented in the results given in Tables (5.5), (5.6) and Tables (5.7), (5.8) respectively, clearly shows an increase of the speedup from 6.232 to 8.695 in experiment 2 (5.777 to 8.772 in experiment 1), and a corresponding increase in efficiency of 33% in experiment 1 and 27% in experiment 2.

On the other hand, by comparing Tables (5.3), (5.4) and Tables (5.7), (5.8) respectively, which represent the experimental results for the static and dynamic techniques using the same number of rules, we can conclude that, the speedup has improved in the two experiments, resulting in an efficiency increase of 7.1% in experiment 1 and 4.6% in experiment 2, when the rules are allocated dynamically.
<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Computing Time (msec)</th>
<th>SPEEDUP</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93600</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>47320</td>
<td>1.978</td>
<td>0.989</td>
</tr>
<tr>
<td>3</td>
<td>31560</td>
<td>2.966</td>
<td>0.989</td>
</tr>
<tr>
<td>4</td>
<td>23810</td>
<td>3.931</td>
<td>0.983</td>
</tr>
<tr>
<td>5</td>
<td>19200</td>
<td>4.875</td>
<td>0.975</td>
</tr>
<tr>
<td>6</td>
<td>16020</td>
<td>5.843</td>
<td>0.974</td>
</tr>
<tr>
<td>7</td>
<td>13760</td>
<td>6.802</td>
<td>0.972</td>
</tr>
<tr>
<td>8</td>
<td>12050</td>
<td>7.768</td>
<td>0.971</td>
</tr>
<tr>
<td>9</td>
<td>10670</td>
<td>8.772</td>
<td>0.975</td>
</tr>
</tbody>
</table>

Table 5.7: Experimental Results for the Parallel Model PFC-DM1. [Rulebase = 350 rules].

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Computing Time (msec)</th>
<th>SPEEDUP</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>118600</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>59860</td>
<td>1.981</td>
<td>0.991</td>
</tr>
<tr>
<td>3</td>
<td>40100</td>
<td>2.958</td>
<td>0.986</td>
</tr>
<tr>
<td>4</td>
<td>30180</td>
<td>3.930</td>
<td>0.982</td>
</tr>
<tr>
<td>5</td>
<td>24160</td>
<td>4.909</td>
<td>0.982</td>
</tr>
<tr>
<td>6</td>
<td>20140</td>
<td>5.889</td>
<td>0.981</td>
</tr>
<tr>
<td>7</td>
<td>17420</td>
<td>6.808</td>
<td>0.973</td>
</tr>
<tr>
<td>8</td>
<td>15320</td>
<td>7.742</td>
<td>0.968</td>
</tr>
<tr>
<td>9</td>
<td>13640</td>
<td>8.695</td>
<td>0.966</td>
</tr>
</tbody>
</table>

Table 5.8: Experimental Results for the Parallel Model PFC-DM1. [Rulebase = 450 rules].
Figure 5.23: The speedup curve for the parallel model PFC-DM1. [Rulebase = 350 rules].

Figure 5.24: The speedup curve for the parallel model PFC-DM1. [Rulebase = 450 rules].
5.4.2 PARALLEL MODELS USING TECHNIQUE 2

In the models described in Section 5.4.1, the matching process is repeated at each cycle of the system run-time. This process of matching the conditions of the rules in $R_i$ (at each cycle) against the elements of DB or LDBs is time-consuming. In this section, all the models described are provided with some control strategies that prevent them from doing this matching in every cycle.

5.4.2.1 The PFC-S2 Model

(Static scheduling with DB retained in the shared memory)

(I) The Data Structures Used

To implement the new control strategies (technique 2), some changes have to be made to the data structures used in technique 1, and the rule structure has been modified. The main data structures used are:

1) The Rulebase:

This is an array of records where each record represents one rule structure. A rule structure is a record with 3 main attributes: The number of premises or conditions (NPrem), the list of premises and the list of conclusions or actions. As an example consider the following rule $r_i$:

$$r_i : \text{IF } C_1 & C_2 & C_3$$

THEN $A_1, A_2$.

This rule is internally represented as a record consisting of:

1) The rule number is equal to $i$.

2) The list of premises is an array with 3 elements: $C_1, C_2, \text{and } C_3$.

3) The list of conclusions is an array with 3 elements: $A_1 \text{ and } A_2$.  

188
In addition, each premise record has a new field called "flag" which indicates, whether the premise (condition) is satisfied or not by the elements of the database, at a given cycle of the consultation. Let \( C_i \) be a premise, we define flag \((C_i)\) as:

\[
\text{flag } (C_i) = \text{TRUE IF } C_i \text{ is satisfied} \\
\text{and flag } (C_i) = \text{FALSE IF } C_i \text{ is not satisfied.}
\]

The conclusion record has also a new field called list-rule. list-rule is an array of Integers, where each element represents a rule number. To understand more the idea of list-rule we shall first give the following definitions:

**Definition 4:**

Let \( r_j \) be a rule:

Premise \((r_j)\) is defined as the set of all the premises in the left-hand side of the rule.

\[
\text{Premise } (r_j) = \{C_\lambda/C_\lambda \text{ is a premise of } r_j\}
\]

\( C_\lambda \in \text{Premise } (r_j) \iff C_\lambda \text{ is a premise of } r_j. \)

In our example here (rule \( r_i \)), we have: \( \text{Premise } (r_i) = \{C_1, C_2, C_3\}. \)

**Definition 5:**

Let \( A_i \) be an action or conclusion, and let \( r_j \) be a rule. We say that the conclusion \( A_i \) affect the rule \( r_j \), if by performing \( A_i \), one or more premise(s) of \( r_j \) will have their flags altered. By performing \( A_i \), the content of the database will be modified; new elements are added, or some existing elements are removed or altered. As a result of this, some conditions will become satisfied and others will be no longer satisfied, and consequently their flags will be altered accordingly.

So, \( A_i \) affect \( r_j \) \( \iff \exists \) at least one condition \( C_\lambda; C_\lambda \in \text{Premise } (r_j) \)

and flag \( (C_\lambda) \) is altered if \( A_i \) is performed.
and, \textit{list-rule} \((A_i) = \{0 \leq j \leq n/r_j \in RB \text{ and } A_i \text{ affect } r_j\}.

In other words, the \textit{list-rule} of a conclusion \(A_i\), stores the numbers of those rules \(r_j\) in the rulebase, which are affected by \(A_i\).

The \textit{list-rule} structure is implemented using an array structure. \textit{list-rule[0]} is used as a counter to keep the number of elements in \textit{list-rule}. As an example, consider the following rules:

\[ r_1 : \text{IF } C_1 \& C_2 \& C_3 \text{ THEN } A_1, A_2, A_3 \]
\[ r_2 : \text{IF } C_4 \& C_5 \text{ THEN } A_4, A_5 \]
\[ r_3 : \text{IF } C_1 \& C_6 \text{ THEN } A_6 \]

We suppose that:

\[ A_1 \text{ affect } r_2 \text{ and } r_3 \]
\[ A_4 \text{ affect } r_1 \]

and

\[ \text{flag}(C_1) = \text{TRUE}, \ \text{flag}(C_2) = \text{FALSE} \]
\[ \text{flag}(C_3) = \text{FALSE}, \ \text{flag}(C_4) = \text{TRUE} \]
\[ \text{flag}(C_5) = \text{FALSE}, \ \text{flag}(C_6) = \text{TRUE} \]

The internal representation of these rules is shown on the next page.

2) The Database:

The same `array of record' structure which was used in technique 1 is also used to implement the database structure in the models discussed in this section.

(II) The Model and its Implementation

For the models described in this section, we assume the following:
- The rules are numbered from 1 to \( n \).
- The processors are numbered from 1 to \( m \).
- The rulebase (RB) is resident in the shared memory of the system.
- The database (DB) is either kept in the shared memory, or copied into the processor's local memories, with one 'reference copy' kept in the shared memory of the system.

The PFC-S2 model consists of 7 main modules (see Figure 5.25). These are:
1. The User-Interface (UI) module

The main job of this module is to communicate with the user (or any other input sources) to collect the facts needed by the system throughout its search for a solution. The collected facts are kept in the database (DB). This module is activated only at the beginning of the system run-time. It is kept separated from the Forward-Chaining inferencing cycles for performance measurement reasons.

2. The Initialize-List-Rules (ILR) module:

This module creates and initializes the list-rule arrays for the conclusion of each rule. This process involves scanning the rulebase (RB) to find those rules which are affected by the conclusion in hand. Once found, the number of the affected rule is added to list-rule, and so on. As a result, the conclusion's list-rules will be filled in with the number of their corresponding affected rules. This module is activated only once at the beginning of the system run-time, as shown in Figure 5.25.

3. The Initialize-Flags (IF) module:

This initializes the "flags" of each premise in the premise part (LHS) of each rule in RB. To do this, the IF module matches the premises against the facts in
DB, and then initializes their flags to TRUE or FALSE, according to the match test result.

4. The Forward-Chaining (FC) module:

This organizes and controls the overall reasoning process. The FC module supervises the last three modules (FAR, SF and UF) and coordinates their activities. It decides when to stop the consultation and give results.

![Diagram of the main modules of PFC-S2](image)

Figure 5.25: The main modules of PFC-S2
5. The Find-Applicable-Rules (FAR) module:

FAR determines the applicability of the rules in the RB and maintains the applicable rules in a set \( S \) together with some other useful information. FAR checks the status of the flags in the premises of each rule and then decides whether or not the rule is applicable. No matching process is needed. The matching process has been replaced with 'checking the flags status' process, which is straightforward, and much less time-consuming.

6. The Select-Fire (SF) module:

This module selects the rules to fire after each cycle, and fires them. When a rule is fired, the actions in its conclusion part are performed. As a result, the content of the DB is modified.

7. The Update-Flags (UF) module:

This module's main task is to update the flags in the premises of the rules which are affected by the conclusions of the fired rules. When a rule \( r_i \) is fired, the actions \((A_j)\)s in its conclusion part are performed. Consequently, the flags of the rules in the list-rules of the \( A_j\)s has to be updated. The algorithm for UF is:

\[
\text{FOR each } r_i \in \text{fired-rules DO}
\]

\[
\text{For each } A_j \in \text{conclusion } (r_i) \text{ DO}
\]

\[
\text{For each rule } r_t \in \text{list-rule } (A_j) \text{ DO}
\]

\[
\text{Update the flags of } r_t \text{ premises.}
\]

\{fired-rules maintains the fired rules at each cycle\}.

(III) Algorithm Structure and Implementation

The PFC-S2 model concept is depicted in Figure 5.26. The model cycles through the following phases:
Figure 5.26: The PFC-S2 concept
Phase 1:

The UI module collects the facts needed by the system throughout the inferencing stages of the consultation. These facts are stored in the DB in shared memory.

Phase 2:

Initialize the list-rule arrays of the rules conclusions. The general algorithm for this phase is:

Step 1: Define the number of processors to be used to execute the job. This number is either defined by the user, or is considered by the system as the default value of (number of CPUs on-line/2). Assume that this number is $q \ (1 \leq q \leq m)$.

Step 2 Set up the $q$ processors for work.

Step 3 Create $q$ processes to carry out the job.

Step 4 Execute the job (Initialize the list-rules) in parallel. This is done according to data-partitioning method with static scheduling of the processes (data = rules).

The static scheduling algorithm simply divides the tasks evenly among the processes. The tasks are scheduled by the processes at run-time, but they are divided in some predetermined way. The algorithm for this step is the following (see Figure 5.27):

1. Determine the number $q$ of processes created for the job.

Cycle 0

2. FOR each process DO:
a) Call 'm-get-myid' to find out the process's ID. (Call this number \( N \))
b) Determine the task to carry out. A task consists of processing \( x \) rules \((x \geq 1)\). In our discussion here \( x \) is considered equal to 1, [task-size= 1]. The task is determined in the following way:

\[
\text{Task} = \text{Process (rule}_I\text{)}
\]

where \( I = N + \lambda q \)

\( \lambda \) is/ 0 \( \leq \lambda \) and \( N + \lambda q \leq n \).

\( n \) is the total number of rules in RB and \( N \) is the ID number of the process.

Note that at cycle 0, \( \lambda = 0 \); at cycle 1, \( \lambda = 1 \) and so on.

c) IF \( I > n \) THEN no work is done and the process becomes idle.

ELSE

- Assign the \( I^{th} \) rule to the process with ID = \( N \)
- Process the rule (rule\(_I\)):

  FOR each conclusion \( A_i \) in the conclusion part of rule\(_I\) DO
  
  - Search the RB for the rules which are affected by \( A_i \),
  
  - IF found, add the rules number to 'list-rule' of \( A_i \).
  
  - Wait for the other \((q - 1)\) processes to finish their tasks.

A more detailed algorithm for this search-finding step is the following:

FOR each rule \( r_t \in \text{RB} \) DO:

  - Check whether or not \( r_t \) is affected by \( A_i \), i.e. check if any of the conditions (premises) of \( r_t \) will have its flag altered if \( A_i \) is performed (becomes satisfied, or the opposite,…).
3. All \( q \) processes have finished their tasks.

4. IF there is more work to be done, i.e. more rules to be processed THEN
   a) Start the next cycle.
   b) Repeat 2 to 4 for the \((n - q)\) rules left unprocessed.

Phase 3:

Initialize the flags in the premises of the rules. The algorithm for this phase is:

Step 1 Define the number of processors to be used to execute the job in parallel. Assume that this number is \( q(1 \leq q \leq m) \).

Step 2 Set up the \( q \) processors for work.

Step 3 Create \( q \) processes to execute the job.
Step 4 Execute the job (Initialize the flags) in parallel. This is done using data-partitioning with the static scheduling technique. The algorithm for this step is:

1. Call 'm-get-numproc' to determine the number of processes created by the system to carry out the job. (This number is \( q \)).

**Cycle 0**

2. **FOR** each process **DO**
   - Call 'm-get-myid' to find out the process's ID (Call this number \( N \)).
   - Determine the process task i.e. figure out which rule to process (task-size = 1). The rule number (\( I \)) is determined as follows:
     \[
     I = N + \lambda q \quad / \quad 0 \leq \lambda \quad \text{and} \quad N + \lambda q \leq n
     \]
     Task = process (rule\( _l \))
     \( \lambda \) is/at cycle 0, \( \lambda = 0 \); at cycle 1, \( \lambda = 1 \); and so on.
   - **IF** \( I > n \) **THEN** no work is done and the process becomes idle.
   **ELSE**
   - Assign rule\( _l \) to the process with ID = \( N \)
   - Process (rule\( _l \)):
     For each condition \( C_l \) in the premise part of rule\( _l \) **DO**
     - Match \( C_l \) against the facts in DB.
     - **IF** \( C_l \) is satisfied **THEN** set the flag of \( C_l \) to TRUE, otherwise set the flag to FALSE.
   - Wait for the other \( (q - 1) \) processes to finish their tasks.

3. All \( q \) processes have finished their tasks.
4. IF there are still more rules to process THEN
   • Start the next cycle
   • Repeat 2 to 4 for the \((n - q)\) rules left unprocessed.

Phase 4:

The concept of this phase is depicted in Figure 5.28. The FC is the master module which controls the other 3 modules and coordinates their activities. The FAR processes the rules in RB, and stores the applicable rules in the set \(S\). Subsequently, FC sends the set \(S\) to SF. In its turn, SF selects a number of rules and fires them. This action changes the content of DB and consequently the flags of the rules’s premises have to be updated accordingly. This updating the flags task is performed by UF. It reads the \textit{list-rules} of the fired-rules’s conclusions and update the flags in the premises of the affected rules, i.e. rules \(r_i / \exists A_j \in \text{one of the fired-rules's conclusion}; i \in \text{list-rule} (A_j)\).

The control is then back to FC, which analyses the results, studies the new status and then decides on whether to start a new cycle or to stop the consultation.

![Figure 5.28: The Concept of Phase 4](image)
This phase can be divided into three sub-phases:

Phase 4a: Find the applicable rules.

Phase 4b: Select and fire rules.

Phase 4c: Update the flags.

**Phase 4a:**

This has the task of finding all the applicable rules at a given cycle of the consultation. It cycles through the following steps:

**Step 1** Define the number of processes to be used to execute the job in parallel. (Call this number \( q \) \((1 \leq q \leq m)\)).

**Step 2** Set up the \( q \) processors for work.

**Step 3** Create \( q \) processes to execute the job.

**Step 4** Execute the job (find applicable rules) in parallel. Again the method used is data-partitioning with static scheduling (data = rules). The algorithm for this step is:

1. Call 'm-get-numproc' to determine the number of processes created by the system (this number is \( q \)).

**Cycle 0**

2. FOR each process DO:
   - Call 'm-get-myid' to determine the process's ID (Call this number \( N \)).
   - Determine my task: find the rule to process (task-size = 1).

   The rule number \( I \) is determined as follows:
   \[
   I = N + \lambda q \quad / \quad 0 \leq \lambda \quad \text{and} \quad N + \lambda q \leq n
   \]

   Task = Process (rule\( I \))

   and \( \lambda \) is/\( \lambda = 0 \) at cycle 0; \( \lambda = 1 \) at cycle 1, etc.
• IF $I > n$ THEN no work is done and the process becomes idle

ELSE

○ Assign rule$_I$ to the process with ID = N.

○ Process rule$_I$:
  
  • Check the conditions of rule$_I$.
  
  • IF all the conditions’s flags of rule$_I$ are set to TRUE $\equiv$
    $$(\forall C_j \in \text{Premise}(\text{rule}_I); \text{flag}(C_j) = \text{TRUE})$$
    THEN
    
    begin if
      
      Lock a lock
      
      Add rule$_I$ to S
      
      Unlock the lock.
    
    end if

○ Wait for the other $(q - 1)$ processes to finish their tasks.

3. All $q$ processes have finished their tasks.

4. IF there are still more rules to process THEN

   • Start the next cycle
   
   • Repeat 2 to 4 for the $(n - q)$ rules left unprocessed.

Phase 4b:

SELECT a number of rules from the set S of applicable rules and FIRE them.

The algorithm for this phase is:

Step 1 Select $\ell$ rules from S. Store these rules in ‘fired-rules’ array.

Step 2 IF $\ell = 1$ THEN fire the rule sequentially.

Step 3 The number of selected rules is greater than one and less than or equal
to the number of allocated processors.
IF \(1 < \ell \leq m\) THEN

1. Allocate \(\ell\) processors, using the ‘m-set-nprocs’ routine.

2. Create \(\ell\) processes. Each process’s task is to fire one rule, i.e. execute the rule’s actions \((A,s)\) and update the DB accordingly. Thus, process 1 fires the rule \(\text{fired-rules}[1]\), process 2 fires \(\text{fired-rules}[2]\), and so on.

3. Each process executes its own task.

**Step 4** The number of selected rules is greater than the number of allocated processors.

ELSE \((\ell > m)\)

1. Create \(m\) processes.

2. FOR each process DO

   - Call ‘m-get-myid’ to determine the process ID. (Call this number \(N\).

   - Call ‘m-get-numprocs’ to determine the number of processes created by the ‘m-fork’ call. (This number is \(m\).

   - Start by processing the \(N^{th}\) element of \(\text{fired-rules}\), i.e. fire the rule \(\text{fired-rule}[N]\).

   - Fire every \(m^{th}\) rule in \(\text{fired-rules}\) until we reach the end of \(\text{fired-rules}\). Thus, fire the rules: \(\text{fired-rules}[N+m]\), \(\text{fired-rules}[N+2m]\), and so on until all the rules in \(\text{fired-rules}\) are processed.

**Phase 4c:**

Update the premise’s flags of the rules in RB following the fire-rule phase (Phase 4b) and the change made to the DB. Assuming that the number of fired
rules is \( \ell \), \( (1 \leq \ell \leq |S|) \), and that the fired rules are kept in the array \textit{fired-rules},

the algorithm for this phase is:

**Step 1**

IF \( \ell = 1 \), i.e. only one rule is fired (call this rule \( r_i \)) THEN

Update the flags in the premises of the rules affected by each of \( r_i \)'s conclusions.

The algorithm for this step is:

FOR each conclusion \( A_k \) in the conclusion part of \( r_i \) DO

\hspace{1cm} FOR every rule \( r_j / r_j \) is affected by \( A_k \) (\( j \in \text{list-rule of } A_k \)) DO

\hspace{2cm} FOR each premise \( C_\lambda \) in the premise part of \( r_j \) DO

\hspace{3cm} Match \( C_\lambda \) against the elements of DB.

\hspace{3cm} IF \( C_\lambda \) is satisfied THEN set \( C_\lambda \)'s flag to TRUE

\hspace{3cm} ELSE set \( C_\lambda \)'s flag to FALSE.

**Step 2**

IF \( (1 < \ell \leq m) \) THEN

1. Allocate \( \ell \) processors, by calling \textit{m-set-nprocs}.

2. Create \( \ell \) processes. Each process's task is to update the flags for the rules in RB affected by the conclusions of one fired rule (see Step 1). Thus, process 1 updates the flags for the rules affected by the conclusions of \textit{fired-rules}[1], process 2 updates the flags for the rules affected by the conclusions of \textit{fired-rules}[2] and so on.

So for each process \( I \), the task \text{Task}I to perform is defined as follows:
• FOR each conclusion \( A_k \) of the rule \( \text{fired-rules} \) \([J]\) DO

• FOR every \( r_j \in \text{RB} / r_j \) is affected by \( A_k \) \((j \in \text{list-rule}(A_k))\) DO

• FOR each premise \( C_\lambda \) of \( r_j \) DO

  • Match \( C_\lambda \) against the elements of \( \text{DB} \).

  • IF \( C_\lambda \) is satisfied THEN set flag to TRUE

  ELSE set flag to FALSE.

3. Each process executes its own task.

Step 3 ELSE \( (\ell > m) \)

1. Create \( m \) processes.

2. FOR each process DO

  • Call 'm-get-myid' to determine the process ID. Call this number \( N \).

  • Call 'm-get-numprocs' to determine the number of processes created by the system to execute the job. (This number is \( m \)).

  • Start by processing the \( N^{th} \) task (Task \( N \)), i.e. update the flags of the rules in \( \text{RB} \) which are affected by the conclusions of the rule \( \text{fired-rules} \) \([N]\). (See Step 2.2 for the task definition).

  • Process every \( m^{th} \) task until we reach the end of \( \text{fired-rules} \).

Thus, process the Tasks: Task \([N + m]\), Task \([N + 2m]\), and so on until all the rules in \( \text{fired-rules} \) are used.

5.4.2.2 Performance Analysis of PFC-S2

Tables 5.9 and 5.10 show the experimental results obtained for the parallel model PFC-S2. In these tables we present the numerical results obtained for three
different modules: IF (Initialize the Flags), ILR (Initialize the List-Rules), and the FC (the Forward Chaining process). These 3 modules are described earlier in this chapter.

The experiments were carried out using two sets of rules with different sizes. These tables are graphically interpreted in Figures (5.29), (5.30) and Figures (5.31), (5.32) respectively.

It can be observed from the figures that the speedup curve follows more closely a log\(_p\) type curve (where \(p\) is the number of processors used); when \(p\) satisfies a small number of processors, (1 to 9). This is due to the fact that the bulk of the processing operations are involved with searching and matching. In addition, extensive rulebase and database acces by the processors, involving communication overheads on the shared memory bus are involved.

It can be observed from the experiments that the efficiency is better when the size of the rulebase is larger. On the other hand, the efficiency index for a small number of processors (1 to 5) indicates an acceptable performance level, however for larger numbers of processors the converse is true. For the FC module, the results show that the use of a small number of processors (1 to 3) indicates an acceptable performance level, whereas for larger number of processors the converse is true. By using 6 or more processors to execute the FC module, the computing time has increased and no speedup is obtained, in contrary the system demonstrates a slow down.

The reason for this are the following:

1. The FC module's main job is to check the rules flags to find the applicable rules, fire them and then update the flags of the rules in the list-rules of the
applicable rules, accordingly. This process is not time consuming and is difficult to parallelize because of the dependancy (data and control) that exist amongst its different constituent parts.

2. The overheads resulting from the partitioning of the FC process amongst the processors, the extensive access to data in the shared memory by each processor, (shared-memory bus contention) and the synchronisation of the concurrent processes have overlapped the computations inside each processor (for \( p > 5 \)).

5.4.2.3 The PFC-SM2 Model

*Static scheduling with local copies of the database kept in each processor's local memory*

(I) The Data Structures Used

The same data structure used for PFC-S2 are again used for PFC-SM2. A main copy of the database called “Reference Copy” (RCDB) is resident in the shared memory of the system. This reference copy is used by the processors to update their local copies of the database (LDB), after every firing of rules.

(II) The Model and its Implementation

The PFC-SM2 model consists of 7 main modules (Figure 5.25). These are the same described before in PFC-S2, with the only difference that for the IF and UF modules the match-test of the premises in the LHS of the rules is conducted against the elements of LDBs. Moreover, the FC module has an additional task, that is to update the local copies LDBs after the firing-rules step (after the Select-Fire phase and before the Update-Flags phase). The model concept is depicted in Figure 5.33.
<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Time(IF)</th>
<th>Speedup (IF)</th>
<th>Efficiency (IF)</th>
<th>Time(ILR)</th>
<th>Speedup (ILR)</th>
<th>Efficiency (ILR)</th>
<th>Time(FC)</th>
<th>Speedup (FC)</th>
<th>Efficiency (FC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26500</td>
<td>1.000</td>
<td>1.000</td>
<td>86320</td>
<td>1.000</td>
<td>1.000</td>
<td>39280</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>13770</td>
<td>1.924</td>
<td>0.962</td>
<td>43630</td>
<td>1.978</td>
<td>0.989</td>
<td>23050</td>
<td>1.704</td>
<td>0.852</td>
</tr>
<tr>
<td>3</td>
<td>9510</td>
<td>2.787</td>
<td>0.929</td>
<td>29900</td>
<td>2.887</td>
<td>0.962</td>
<td>17500</td>
<td>2.245</td>
<td>0.748</td>
</tr>
<tr>
<td>4</td>
<td>7680</td>
<td>3.451</td>
<td>0.863</td>
<td>23830</td>
<td>3.622</td>
<td>0.906</td>
<td>15440</td>
<td>2.544</td>
<td>0.636</td>
</tr>
<tr>
<td>5</td>
<td>6420</td>
<td>4.128</td>
<td>0.826</td>
<td>19700</td>
<td>4.382</td>
<td>0.876</td>
<td>14720</td>
<td>2.668</td>
<td>0.534</td>
</tr>
<tr>
<td>6</td>
<td>6090</td>
<td>4.351</td>
<td>0.725</td>
<td>17860</td>
<td>4.833</td>
<td>0.806</td>
<td>16310</td>
<td>2.408</td>
<td>0.401</td>
</tr>
<tr>
<td>7</td>
<td>5660</td>
<td>4.682</td>
<td>0.669</td>
<td>16700</td>
<td>5.169</td>
<td>0.738</td>
<td>16800</td>
<td>2.338</td>
<td>0.334</td>
</tr>
<tr>
<td>8</td>
<td>5360</td>
<td>4.944</td>
<td>0.618</td>
<td>16180</td>
<td>5.335</td>
<td>0.667</td>
<td>17010</td>
<td>2.309</td>
<td>0.289</td>
</tr>
<tr>
<td>9</td>
<td>5180</td>
<td>5.116</td>
<td>0.568</td>
<td>16010</td>
<td>5.392</td>
<td>0.599</td>
<td>17060</td>
<td>2.302</td>
<td>0.256</td>
</tr>
</tbody>
</table>

Table 5.9: Experimental Results for different modules of the Parallel Model PFC-S2. [RuleBase = 350 rules].
<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Time(IF) (msec)</th>
<th>Speedup (IF)</th>
<th>Efficiency (IF)</th>
<th>Time(ILR) (msec)</th>
<th>Speedup (ILR)</th>
<th>Efficiency (ILR)</th>
<th>Time(FC) (msec)</th>
<th>Speedup (FC)</th>
<th>Efficiency (FC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35560</td>
<td>1.000</td>
<td>1.000</td>
<td>140500</td>
<td>1.000</td>
<td>1.000</td>
<td>50120</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>18630</td>
<td>1.909</td>
<td>0.954</td>
<td>72330</td>
<td>1.942</td>
<td>0.971</td>
<td>28540</td>
<td>1.756</td>
<td>0.878</td>
</tr>
<tr>
<td>3</td>
<td>12870</td>
<td>2.763</td>
<td>0.921</td>
<td>49020</td>
<td>2.866</td>
<td>0.955</td>
<td>22260</td>
<td>2.252</td>
<td>0.751</td>
</tr>
<tr>
<td>4</td>
<td>10080</td>
<td>3.528</td>
<td>0.882</td>
<td>38120</td>
<td>3.686</td>
<td>0.921</td>
<td>19060</td>
<td>2.630</td>
<td>0.657</td>
</tr>
<tr>
<td>5</td>
<td>8420</td>
<td>4.223</td>
<td>0.845</td>
<td>31590</td>
<td>4.448</td>
<td>0.890</td>
<td>17900</td>
<td>2.800</td>
<td>0.560</td>
</tr>
<tr>
<td>6</td>
<td>7470</td>
<td>4.760</td>
<td>0.793</td>
<td>27880</td>
<td>5.039</td>
<td>0.840</td>
<td>17610</td>
<td>2.846</td>
<td>0.474</td>
</tr>
<tr>
<td>7</td>
<td>6870</td>
<td>5.176</td>
<td>0.739</td>
<td>25790</td>
<td>5.448</td>
<td>0.778</td>
<td>17540</td>
<td>2.857</td>
<td>0.408</td>
</tr>
<tr>
<td>8</td>
<td>6410</td>
<td>5.548</td>
<td>0.693</td>
<td>24820</td>
<td>5.661</td>
<td>0.708</td>
<td>17870</td>
<td>2.805</td>
<td>0.351</td>
</tr>
<tr>
<td>9</td>
<td>6050</td>
<td>5.878</td>
<td>0.653</td>
<td>24380</td>
<td>5.763</td>
<td>0.640</td>
<td>18340</td>
<td>2.733</td>
<td>0.304</td>
</tr>
</tbody>
</table>

Table 5.10: Experimental Results for different modules of the Parallel Model PFC-S2. [RuleBase = 450 rules].
Figure 5.29: The speedup curve for the IF module in the parallel model PFC-S2. 
[Rulebase = 350 rules].

Figure 5.30: The speedup curve for the IF module in the parallel model PFC-S2. 
[Rulebase = 450 rules].
Figure 5.31: The speedup curve for the ILR module in the parallel model PFC-S2.  
[Rulebase = 350 rules].

Figure 5.32: The speedup curve for the ILR module in the parallel model PFC-S2.  
[Rulebase = 450 rules].
Figure 5.33: The PFC-SM2 Concept
5.4.2.4 Performance Analysis of PFC-SM2

The experimental results obtained for the parallel model PFC-SM2 are shown in Tables 5.11 and 5.12. The graphical interpretations of these tables are shown in Figures (5.34), (5.35) and Figures (5.36), (5.37) respectively. Again, the experiments were carried out using two sets of rules with different sizes. As for PFC-S2, it can be concluded from the two tables that the results (speedup and efficiency) have improved for an increasing number of rules. A comparison between the use of shared memory and local memories for the IF module in PFC-S2 and PFC-SM2 systems, which is presented in the results given in Tables (5.9), (5.10) and Tables (5.11), (5.12) respectively, clearly shows the speedup to increase from 5.878 to 8.829 in experiment 2 (5.116 to 8.676 in experiment 1) and a corresponding increase in efficiency of more than 30% for the two experiments.

For the two other modules ILR and FC, a similar conclusion to that of PFC-S2 can be drawn.

5.4.2.5 The PFC-D2 Model
(Dynamic Scheduling with DB retained in the shared Memory)
(I) The Data Structure

PFC-D2 uses the same data structure used for PFC-S2. These are the RuleBase RB, the DataBase DB, the set of applicable rules S and the set of fired rules (fired-rules).

(II) The Model and its Implementation

The model consists of 7 main modules (see Figure 5.25). These are:

1. The User-Interface (UI)
<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Time(IF) msec</th>
<th>Speedup (IF)</th>
<th>Efficiency (IF)</th>
<th>Time(ILR) msec</th>
<th>Speedup (ILR)</th>
<th>Efficiency (ILR)</th>
<th>Time(FC) msec</th>
<th>Speedup (FC)</th>
<th>Efficiency (FC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26550</td>
<td>1.000</td>
<td>1.000</td>
<td>85470</td>
<td>1.000</td>
<td>1.000</td>
<td>39920</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>13370</td>
<td>1.986</td>
<td>0.993</td>
<td>44450</td>
<td>1.923</td>
<td>0.961</td>
<td>23060</td>
<td>1.731</td>
<td>0.866</td>
</tr>
<tr>
<td>3</td>
<td>8990</td>
<td>2.953</td>
<td>0.984</td>
<td>29920</td>
<td>2.857</td>
<td>0.952</td>
<td>18030</td>
<td>2.214</td>
<td>0.738</td>
</tr>
<tr>
<td>4</td>
<td>6810</td>
<td>3.899</td>
<td>0.975</td>
<td>23220</td>
<td>3.681</td>
<td>0.920</td>
<td>16020</td>
<td>2.492</td>
<td>0.623</td>
</tr>
<tr>
<td>5</td>
<td>5390</td>
<td>4.926</td>
<td>0.985</td>
<td>19650</td>
<td>4.350</td>
<td>0.870</td>
<td>15450</td>
<td>2.584</td>
<td>0.517</td>
</tr>
<tr>
<td>6</td>
<td>4560</td>
<td>5.822</td>
<td>0.970</td>
<td>17690</td>
<td>4.832</td>
<td>0.805</td>
<td>15590</td>
<td>2.561</td>
<td>0.427</td>
</tr>
<tr>
<td>7</td>
<td>3870</td>
<td>6.860</td>
<td>0.980</td>
<td>16550</td>
<td>5.164</td>
<td>0.738</td>
<td>16150</td>
<td>2.472</td>
<td>0.353</td>
</tr>
<tr>
<td>8</td>
<td>3410</td>
<td>7.786</td>
<td>0.973</td>
<td>16140</td>
<td>5.296</td>
<td>0.662</td>
<td>17040</td>
<td>2.343</td>
<td>0.293</td>
</tr>
<tr>
<td>9</td>
<td>3060</td>
<td>8.676</td>
<td>0.964</td>
<td>15870</td>
<td>5.386</td>
<td>0.598</td>
<td>17730</td>
<td>2.252</td>
<td>0.250</td>
</tr>
</tbody>
</table>

Table 5.11: Experimental Results for different modules of the Parallel Model PFC-SM2. [RuleBase = 350 rules].
### Table 5.12: Experimental Results for different modules of the Parallel Model PFC-SM2. [RuleBase = 450 rules].

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Time(IF) [msec]</th>
<th>Speedup (IF)</th>
<th>Efficiency (IF)</th>
<th>Time(ILR) [msec]</th>
<th>Speedup (ILR)</th>
<th>Efficiency (ILR)</th>
<th>Time(FC) [msec]</th>
<th>Speedup (FC)</th>
<th>Efficiency (FC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35670</td>
<td>1.000</td>
<td>1.000</td>
<td>140630</td>
<td>1.000</td>
<td>1.000</td>
<td>51020</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>18210</td>
<td>1.959</td>
<td>0.979</td>
<td>73830</td>
<td>1.905</td>
<td>0.952</td>
<td>29120</td>
<td>1.752</td>
<td>0.876</td>
</tr>
<tr>
<td>3</td>
<td>11950</td>
<td>2.985</td>
<td>0.995</td>
<td>49040</td>
<td>2.868</td>
<td>0.956</td>
<td>22040</td>
<td>2.315</td>
<td>0.772</td>
</tr>
<tr>
<td>4</td>
<td>9050</td>
<td>3.941</td>
<td>0.985</td>
<td>37380</td>
<td>3.762</td>
<td>0.941</td>
<td>18770</td>
<td>2.718</td>
<td>0.680</td>
</tr>
<tr>
<td>5</td>
<td>7270</td>
<td>4.906</td>
<td>0.981</td>
<td>31110</td>
<td>4.520</td>
<td>0.904</td>
<td>17360</td>
<td>2.939</td>
<td>0.588</td>
</tr>
<tr>
<td>6</td>
<td>6100</td>
<td>5.848</td>
<td>0.975</td>
<td>27840</td>
<td>5.051</td>
<td>0.842</td>
<td>17300</td>
<td>2.949</td>
<td>0.492</td>
</tr>
<tr>
<td>7</td>
<td>5230</td>
<td>6.820</td>
<td>0.974</td>
<td>26690</td>
<td>5.269</td>
<td>0.753</td>
<td>18080</td>
<td>2.822</td>
<td>0.403</td>
</tr>
<tr>
<td>8</td>
<td>4570</td>
<td>7.805</td>
<td>0.976</td>
<td>24850</td>
<td>5.659</td>
<td>0.707</td>
<td>18250</td>
<td>2.796</td>
<td>0.349</td>
</tr>
<tr>
<td>9</td>
<td>4040</td>
<td>8.829</td>
<td>0.981</td>
<td>24380</td>
<td>5.768</td>
<td>0.641</td>
<td>18710</td>
<td>2.727</td>
<td>0.303</td>
</tr>
</tbody>
</table>
Figure 5.34: The speedup curve for the IF module in the parallel model PFC-SM2. [Rulebase = 350 rules].

Figure 5.35: The speedup curve for the IF module in the parallel model PFC-SM2. [Rulebase = 450 rules].
Figure 5.36: The speedup curve for the ILR module in the parallel model PFC-SM2. 
[Rulebase = 350 rules].

Figure 5.37: The speedup curve for the ILR module in the parallel model PFC-SM2. 
[Rulebase = 450 rules].
2. The Initialize-List-Rules (ILR)
3. The Initialize-Flags (IF)
4. The Forward-Chaining (FC)
5. The Find-Applicable-Rules (FAR)
6. The Select-Fire (SF)
7. The Update-Flags (UF)

These modules are described in Section 5.4.2.1.

(III) Algorithm Structure and Implementation

Definition 6:

The variable $BITE$ is used to refer to the number of tasks to be assigned to one processor at one time i.e. after each job request the processor makes.

$$BITE \equiv$$

The number $n_1$ of tasks allocated to one processor at each job – request made by this processor.

Note 1:

In our discussion here, the number $n_1$ is considered as equal to 1.

$$BITE = \text{one task} = 1$$

Definition 7:

A task consists of a number $n_2$ of rules to process. This number $n_2$ is called the “task-size”.

$$\text{Task-size} \equiv \text{number of rules in the task} = n_2.$$
Note 2: In our discussion here, this number \( n_2 \) is considered as equal to 1. \( (n_2 = 1) \).

Task-size = one rule = 1

**Definition 8:**
Consider a task \( T \) which consists of investigating \( k \) rules i.e. task-size \( (T) = k \):

\[ T \equiv \text{investigate} (\text{rule}_{I_1}, \text{rule}_{I_2}, \ldots, \text{rule}_{I_k}). \]

We define the two variables \( \text{Base} \) and \( \text{Top} \) for the task \( T \) as:

\[ \text{Base} (T) = I_1 \]
\[ \text{Top} (T) = I_k \]

In other words, the \( \text{Base} \) of the task \( T \) is equal to the index of the first rule in \( T \) (\( \text{rule}_{I_1} \)). The \( \text{Top} \) of the task \( T \) is equal to the index of the last rule in \( T \) (\( \text{rule}_{I_k} \)).

Processing the task \( T \) is defined as:

- **Process** \( T \equiv \)
  
  \[ \text{FOR } I = \text{Base} \text{ TO } \text{Top} \text{ DO} \]
  
  Process (\( \text{rule}_I \)).

**Note 3:** Since the task-size of a task \( T \) is considered to be equal to 1 in the discussion here, thus:

\[ T \equiv \text{investigate}(\text{rule}_I) \ \{ \text{one rule} \} \]

and

\[ \text{Base}(T) = I = \text{Top}(T). \]

The tasks are arranged in a **task-queue**. Each processor removes one BITE from the queue (one task) and processes it, i.e. processes the rules in the task. The scheduling algorithm includes the following steps:
1. Lock a lock.

2. Check the task-queue and verify that there is still work to be done.

3. Remove a task from the task-queue.

4. Unlock the lock.

5. Process the task.

6. Repeat step 1 through 5 until all the work is finished.

The PFC-D2 model concept is depicted in Figure 5.38. The model cycles through the following phases:

**Phase 1:**

The UI module collects the facts needed by the system throughout the consultation. These facts are stored in DB in the shared memory of the system.

**Phase 2:**

Initialize the *list-rule* arrays of the rules conclusions. The algorithm for this phase is the following:

**Step 1** Define the number of processors to execute the job in parallel. This value is either defined by the user, or is considered by the system as the default value of (number of CPUs on-line/2). Assume that this number is $q(1 \leq q \leq m)$.

**Step 2** Set up the $q$ processors for work.

**Step 3** Create $q$ processes to carry out the job.

**Step 4** Execute the job (Initialize the *list-rules*) in parallel. This is done according to the data-partitioning method with dynamic scheduling (data = rules). The algorithm for this step is:
1. Define the variables Bite and task-size. These are considered as equal to 1 in the discussion here. (Bite = one task = 1; task-size = 1).

2. FOR each process DO:

   a. Call 'm-get-numprocs' to find out the number of processes created for the job (this number is q).

   b. Call 'm-next' to find out which task (rule) to process. The routine m-next increments a shared global counter and returns an integer value. The first call to m-next returns the value 1, the second call returns 2, and so on. Assume that the returned value is equal to λ.

   c. Determine the task $T_\lambda$ to perform. This is done in the following way:

      $T_\lambda$ is / Base = Bite $\times$ m-next

      and Top = Base + Bite - 1

      Thus, with Bite = 1 and m-next = λ we obtain:

      Base = 1 $\times$ λ = λ

      Top = λ + 1 - 1 = λ

      and

      $T_\lambda$ is / Base ($T_\lambda$) = λ

      Top ($T_\lambda$) = λ

   d. Process the task $T_\lambda$:

      FOR $I = Base(T_\lambda)$ TO Top($T_\lambda$) DO

      Process (rule$\_I$).

      By replacing Base and Top with their values we will have:

      FOR $I = \lambda$ TO $\lambda$ DO ≡ Process(rule$\_\lambda$).

      Process (rule$\_\lambda$)

      The algorithm for this step is:

      • IF $\lambda > n$ THEN no work is done and the process becomes idle.

221
"ELSE"

- Process the rule rule\(_\lambda\):

  FOR each conclusion \(A_i\) in the conclusion part of rule\(_\lambda\) DO
Search the RB for the rules which are affected by \( A_i \).

IF found, add the rules numbers to \( \text{list-rule}(A_i) \)

A more detailed algorithm for this search-found step is the following:

FOR each rule \( r_t \in \text{RB} \) DO

- Check whether or not, \( r_t \) is affected by \( A_i \), i.e. check if any of the conditions (premises) of \( r_t \) will have its flag altered if \( A_i \) is performed (becomes satisfied, or the opposite, . . ).

- IF \( r_t \) is affected by \( A_i \) THEN add \( r_t \) to \( \text{list-rule} \) of \( A_i \)

e. IF there is more work to do i.e. more tasks in the \( \text{task-queue} \) THEN

GOTO Step 4(2.a).

Step 5 Terminate the processes.

Phase 3:

Initialize the flags in the premises of the rules. The algorithm for this phase is:

Step 1 Define the number of processors to execute the job. Assume that this number is \( q (1 \leq q \leq m) \).

Step 2 Set up the \( q \) processors for work.

Step 3 Create \( q \) processes to carry out the job.

Step 4 Execute the job (initialize the flags) in parallel. This is done using a data-partitioning with dynamic scheduling technique. The algorithm for this step is as follows:

1. Define the variables \( B_t \) and \( \text{task-size} \). These are considered as equal to 1 in the discussion here.
2. FOR each process DO:
   a. Call 'm-get-numprocs' to find out the number of processes created for the job. (This number is q).
   b. Call 'm-next' to find out which task to process. This call returns an integer value \( \lambda \) \((\lambda \geq 1)\).
   c. Determine the task \( T_\lambda \) to carry out:
      \[
      T_\lambda \text{ is } / \quad \text{Base} (T_\lambda) = \text{Bite} \times m-\text{next} = \lambda \\
      \text{Top} (T_\lambda) = \text{Base} + \text{Bite} - 1 = \lambda + 1 - 1 = \lambda
      \]
   d. Process the task \( T_\lambda \), i.e. process rule_\lambda: The algorithm for this step is:
      - IF \( \lambda > n \) THEN no work is done and the process becomes idle.
      - ELSE
        o Process the rule 'rule_\lambda':
          FOR each condition \( C_\ell \) in the premise part of rule_\lambda DO
            - Match \( C_\ell \) against the facts in DB.
            - IF \( C_\ell \) is satisfied THEN set the flag of \( C_\ell \) to TRUE, otherwise, set the flag to FALSE.
        e. IF there is more work to do i.e. more tasks in the 'task-queue' THEN
          GOTO Step 4(2.a).

Step 5 Terminate the processes.

Phase 4:

The concept of this phase is depicted in Figure 5.28. The FC is the 'Master Module', which controls the other 3 modules and coordinate their activities. The FAR investigates the rules in RB, finds those applicable rules and stores them in
the set $S$ in the shared memory. The FC sends then the set $S$ to SF, which selects a number of rules from $S$ and fire them. The rules selected for firing are maintained in the array ‘fired-rules’ (the rules numbers). After firing the selected rules, the content of the DB is modified, and consequently the flags of the rules’s premises have to be updated accordingly. This task of updating the premises flags is performed by UF. It reads the ‘list-rule’ of each rule in ‘fired-rules’, and update the flags of the premises in the affected rules, i.e. update the flags of the rules $r_i/\exists A_j \in$ one of the fired rules conclusion; $i \in list-rule (A_j)$.

This is explained in detail later in phase 4c.

The control is then back to FC, which analyses the results, and decides on whether to start a new cycle or to stop the consultation.

This phase can be divided into three sub-phases:

**Phase 4a:** Find the applicable rules.

**Phase 4b:** Select and fire rules.

**Phase 4c:** Update the flags.

**Phase 4a:**

This finds the applicable rules at a given cycle of the consultation. It cycles through the following steps:

**Step 1** Define the number of processors to be used to execute the job in parallel. (Call this number $q/1 \leq q \leq m$).

**Step 2** Set up the $q$ processors for work.

**Step 3** Create $q$ processes to execute the job.
Step 4 Execute the job (Find applicable rules) in parallel. The algorithm for this step is:

1. Define the variables Bite and task-size. These are considered as equal to 1 in the discussion here.

2. FOR each process DO:

   a. Call ‘m-get-numprocs’ to find out the number of processes created for the jobs. (This number is q).

   b. Call ‘m-next’ to determine the task to process. This call returns a value λ. (λ ≥ 1)

   c. Determine the task $T_\lambda$ to carry out:

   $T_\lambda$ is / Base ($T_\lambda$) = Bite * m-next = λ

   Top ($T_\lambda$) = Base + Bite − 1 = λ.

   d. Process the task $T_\lambda$, i.e. process $rule_\lambda$. The algorithm for this step is:

      • IF $\lambda > n$ THEN no work is done and the process becomes idle.
      • ELSE

         o Process the rule ‘$rule_\lambda$’:

            • Check the conditions of ($rule_\lambda$).

            • IF all the conditions’s flags of $rule_\lambda$ are set to TRUE

               $\equiv (\forall C_j \in \text{Premise (}rule_\lambda\text{)}; \text{flag (}C_j\text{)} = \text{TRUE})$ THEN

               • Lock a lock.

               • Add $rule_\lambda$ to S.

               • Unlock the lock.
e. IF there is more work to do, i.e. more tasks in the 'task-queue' THEN

GOTO Step 4(2.a).

Step 5 Terminate the processes with a call to 'm-kill'.

Phase 4b:

SELECT a number of rules from the set $S$ of applicable rules and FIRE them.

The algorithm for this phase is:

Step 1 Select $\ell$ rules from $S$. Store these rules in the array ‘fire-rules’.

Step 2 IF $\ell = 1$ THEN fire the rule sequentially.

Step 3 The number of selected rules are greater than one.

IF $(1 < \ell \leq m)$ THEN

1. Allocate $\ell$ processors.

2. Create $\ell$ processes. Each process's task is to fire one rule, i.e. execute
the rule's actions and update DB accordingly. Thus process 1 fires
the rule $\text{fired-rules}[1]$, process 2 fires $\text{fired-rules}[2]$ and so on.

3. Each process executes its own task.

Step 4 The number of selected rules is greater than the number of allocated
processors.

ELSE ($\ell > m$)

1. Create $m$ processes to carry out the job.

2. Define the variables ‘Bite’ and ‘task-size’. (Bite = 1, task-size = 1).

3. FOR each process DO:

   a) Call ‘m-get-numprocs’ to determine the number of processes cre­
       ated for the job. (This number is $m$).
b) Call ‘m-next’. This returns a value $\lambda$, ($\lambda \geq 1$)

c) IF ($\lambda > |\text{fired-rules}|$) THEN no work is done and the process becomes idle.

d) ELSE: Process the $\lambda^{th}$ element of $\text{fired-rules}$, i.e. fire $\text{fired}\!\!\!\!\!\!”\text{rules} [\lambda]$
and update DB accordingly.

e) IF there are more rules in $\text{fired-rules}$ THEN GOTO Step 4(3.a).

Step 5 Terminate the processes.

Phase 4c:

Update the premises's flags of the rules in RB, after the changes made to the DB, resulting from the rules firing in phase 4b. Assuming that the number of the fired rules is $\ell$ ($1 \leq \ell \leq |S|$), the algorithm for this phase is:

Step 1

IF $\ell = 1$, i.e. only one rule is fired (Call this rule $r_i$) THEN

- Update the flags in the premises of the rules affected by each of $r_i$'s conclusions.

The algorithm for this step is:

FOR each conclusion $A_k$ in the conclusion part of $r_i$ DO

FOR every rule $r_j/r_j$ is affected by $A_k$ ($j \in \text{list-rules of } A_k$) DO

FOR each premise $C_\lambda$ in the premise part of $r_j$ DO

- Match $C_\lambda$ against the elements of DB.

- IF $C_\lambda$ is satisfied THEN set $C_\lambda$'s flag to TRUE

- ELSE set $C_\lambda$'s flag to FALSE.
Step 2

\[
\text{IF } (1 < \ell \leq m) \text{ THEN}
\]

1. Allocate \(\ell\) processors for the job.

2. Create \(\ell\) processes. Each process's job is to update the flags for the rules in RB affected by the conclusions of one fired rule, (see step 1). Thus, process 1 updates the flags for the rules affected by the conclusions of \textit{fired-rules} [1]. Process 2 updates the flags for the rules affected by the conclusions of \textit{fired-rules} [2], and so on.

So for each process \(I\), the task TaskI is:

\[
\text{FOR each conclusion } A_k \text{ of the rule } \textit{fired-rules} [I] \text{ DO}
\]

\[
\text{FOR each premise } C_\lambda \text{ of } r_j \text{ DO}
\]

\[
\quad \cdot \text{Match } C_\lambda \text{ against the elements of DB.}
\]

\[
\quad \cdot \text{IF } C_\lambda \text{ is satisfied THEN set } C_\lambda \text{'s flag to TRUE}
\]

\[
\text{ELSE set } C_\lambda \text{'s flag to FALSE.}
\]

3. Each process executes its own task.

Step 3

\[
\text{ELSE } (\ell > m)
\]

1. Create \(m\) processes.

2. Define \textit{Bite} and \textit{task-size}. (\(\textit{Bite} = 1; \textit{task-size} = 1\)).

3. FOR each process DO:

   a. Call 'm-get-numprocs' to determine the number of processes created. (This number is \(m\)).

   b. Call 'm-next'. This returns a value \(\lambda\) (\(\lambda \geq 1\)).
c. IF ($\lambda > |\text{fired-rules}|$) THEN no work is done and the process becomes idle.

d. Process the $\lambda^{th}$ task ($\text{Task}_\lambda$), i.e. update the flags of the rules in RB, which are affected by the conclusions of the rule $\text{fired-rules} [\lambda]$ (see Step 2.2 for the algorithm).

e. IF there are more tasks in the task-queue, i.e. more rules in fired-rules THEN GOTO Step 3(3.a).

Step 4 Terminate the processes.

5.4.2.6 Performance Analysis of PFC-D2

Tables 5.13 and 5.14 show the experimental results obtained for the parallel model PFC-D2. Again, the results shown are for the three modules IF, ILR and FC. The graphical interpretations of the tables are shown in Figures (5.39), (5.40) and (5.41), (5.42) respectively. Two sets of experiments were carried out using two sets of rules with different sizes.

It can be observed from the experiments that the results (speedup and efficiency) show a slight improvement for an increasing number of rules.

By comparing Tables (5.9) and (5.10) with Tables (5.13) and (5.14), respectively, which represent the timing results for the static and dynamic techniques using the same number of rules, it can be observed that the speedup has generally improved, resulting in an efficiency increase, when the rules are dynamically allocated.

For the FC module, the same remarks given before for the PFC-S2 system are still valid for PFC-D2. Nevertheless, the results obtained when using the dynamic scheduling technique are better than those obtained using static scheduling.
<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Time(IF) (msec)</th>
<th>Speedup (IF)</th>
<th>Efficiency (IF)</th>
<th>Time(ILR) (msec)</th>
<th>Speedup (ILR)</th>
<th>Efficiency (ILR)</th>
<th>Time(FC) (msec)</th>
<th>Speedup (FC)</th>
<th>Efficiency (FC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26520</td>
<td>1.000</td>
<td>1.000</td>
<td>86530</td>
<td>1.000</td>
<td>1.000</td>
<td>39470</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>13800</td>
<td>1.922</td>
<td>0.961</td>
<td>43790</td>
<td>1.976</td>
<td>0.988</td>
<td>22780</td>
<td>1.733</td>
<td>0.866</td>
</tr>
<tr>
<td>3</td>
<td>9660</td>
<td>2.745</td>
<td>0.915</td>
<td>29730</td>
<td>2.911</td>
<td>0.970</td>
<td>18030</td>
<td>2.189</td>
<td>0.730</td>
</tr>
<tr>
<td>4</td>
<td>7600</td>
<td>3.489</td>
<td>0.872</td>
<td>23280</td>
<td>3.717</td>
<td>0.929</td>
<td>15790</td>
<td>2.500</td>
<td>0.625</td>
</tr>
<tr>
<td>5</td>
<td>6590</td>
<td>4.024</td>
<td>0.805</td>
<td>19630</td>
<td>4.408</td>
<td>0.882</td>
<td>15280</td>
<td>2.583</td>
<td>0.517</td>
</tr>
<tr>
<td>6</td>
<td>5900</td>
<td>4.495</td>
<td>0.749</td>
<td>17560</td>
<td>4.928</td>
<td>0.821</td>
<td>15400</td>
<td>2.563</td>
<td>0.427</td>
</tr>
<tr>
<td>7</td>
<td>5510</td>
<td>4.813</td>
<td>0.688</td>
<td>16390</td>
<td>5.279</td>
<td>0.754</td>
<td>15800</td>
<td>2.498</td>
<td>0.357</td>
</tr>
<tr>
<td>8</td>
<td>5240</td>
<td>5.061</td>
<td>0.633</td>
<td>15910</td>
<td>5.439</td>
<td>0.680</td>
<td>16350</td>
<td>2.414</td>
<td>0.302</td>
</tr>
<tr>
<td>9</td>
<td>5040</td>
<td>5.262</td>
<td>0.585</td>
<td>15540</td>
<td>5.568</td>
<td>0.619</td>
<td>16920</td>
<td>2.333</td>
<td>0.259</td>
</tr>
</tbody>
</table>

Table 5.13: Experimental Results for different modules of the Parallel Model PFC-D2. [RuleBase = 350 rules]
<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Time(IF) msec</th>
<th>Speedup (IF)</th>
<th>Efficiency (IF)</th>
<th>Time(ILR) msec</th>
<th>Speedup (ILR)</th>
<th>Efficiency (ILR)</th>
<th>Time(FC) msec</th>
<th>Speedup (FC)</th>
<th>Efficiency (FC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35590</td>
<td>1.000</td>
<td>1.000</td>
<td>140760</td>
<td>1.000</td>
<td>1.000</td>
<td>50320</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>18580</td>
<td>1.916</td>
<td>0.958</td>
<td>72770</td>
<td>1.934</td>
<td>0.967</td>
<td>28830</td>
<td>1.745</td>
<td>0.873</td>
</tr>
<tr>
<td>3</td>
<td>12700</td>
<td>2.802</td>
<td>0.934</td>
<td>48610</td>
<td>2.896</td>
<td>0.965</td>
<td>21360</td>
<td>2.356</td>
<td>0.785</td>
</tr>
<tr>
<td>4</td>
<td>9990</td>
<td>3.563</td>
<td>0.891</td>
<td>38080</td>
<td>3.696</td>
<td>0.924</td>
<td>18780</td>
<td>2.679</td>
<td>0.670</td>
</tr>
<tr>
<td>5</td>
<td>8420</td>
<td>4.227</td>
<td>0.845</td>
<td>31190</td>
<td>4.513</td>
<td>0.903</td>
<td>17570</td>
<td>2.864</td>
<td>0.573</td>
</tr>
<tr>
<td>6</td>
<td>7450</td>
<td>4.777</td>
<td>0.796</td>
<td>27780</td>
<td>5.067</td>
<td>0.844</td>
<td>17320</td>
<td>2.905</td>
<td>0.484</td>
</tr>
<tr>
<td>7</td>
<td>6790</td>
<td>5.242</td>
<td>0.749</td>
<td>25600</td>
<td>5.498</td>
<td>0.785</td>
<td>17400</td>
<td>2.892</td>
<td>0.413</td>
</tr>
<tr>
<td>8</td>
<td>6370</td>
<td>5.587</td>
<td>0.698</td>
<td>24580</td>
<td>5.727</td>
<td>0.716</td>
<td>17810</td>
<td>2.825</td>
<td>0.353</td>
</tr>
<tr>
<td>9</td>
<td>6030</td>
<td>5.902</td>
<td>0.656</td>
<td>24050</td>
<td>5.853</td>
<td>0.650</td>
<td>18250</td>
<td>2.757</td>
<td>0.306</td>
</tr>
</tbody>
</table>

Table 5.14: Experimental Results for different modules of the Parallel Model PFC-D2. [RuleBase = 450 rules].
Figure 5.39: The speedup curve for the IF module in the parallel model PFC-D2. 
*Rulebase = 350 rules*. 

Figure 5.40: The speedup curve for the IF module in the parallel model PFC-D2. 
*Rulebase = 450 rules*. 

233
Figure 5.41: The speedup curve for the ILR module in the parallel model PFC-D2. 
[Rulebase = 350 rules].

Figure 5.42: The speedup curve for the ILR module in the parallel model PFC-D2. 
[Rulebase = 450 rules].
5.4.2.7 The PFC-DM2 Model

(Dynamic scheduling with local copies of the database kept in each processor's local memory)

(I) The Data Structures Used

The PFC-DM2 model uses the same data structures used for PFC-D2. A main copy of the database called “Reference Copy” (RCDB) is resident in the shared memory of the system. This reference copy is used by the processors to update their local copies of the database (LDB)s, after every alteration of RCDB.

(II) The Model and its Implementation

The PFC-DM2 model consists of 7 main modules (see Figure 5.25). These are the same described before in PFC-D2, with the only difference that for the IF and UF modules the match-test of the rules conditions is conducted against the elements of LDBs. In addition, the FC module has an extra-task, that is to update the local copies LDBs after the firing-rules step (after the Select-Fire phase and before the Update-Flags phase).

5.4.2.8 Performance Analysis of PFC-DM2

Tables 5.15 and 5.16 show the experimental results obtained for the three modules: IF, ILR and FC for the parallel model PFC-DM2. Their graphical interpretations are shown in Figures (5.43), (5.44) and Figures (5.45), (5.46) respectively.

As before, two sets of experiments using different numbers of rules, were carried out.

Again, it can be concluded from the two tables that the results show a slight improvement for an increasing number of rules.
The comparison between the use of shared memory and local memories for the IF module in the PFC-D2 and PFC-DM2 systems, as presented by the results given in Tables (5.13), (5.14) and Tables (5.15), (5.16) respectively, clearly shows an increase in the speedup from 5.262 to 8.789 in experiment 1 (5.902 to 8.791 in experiment 2) and a corresponding increase in efficiency of 39.2% in experiment 1 (32.1% in experiment 2).

Finally, for the two remaining modules ILR and FC, similar conclusions to that of PFC-SM2 can be drawn.
<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Time(IF) msec</th>
<th>Speedup (IF)</th>
<th>Efficiency (IF)</th>
<th>Time(ILR) msec</th>
<th>Speedup (ILR)</th>
<th>Efficiency (ILR)</th>
<th>Time(FC) msec</th>
<th>Speedup (FC)</th>
<th>Efficiency (FC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26280</td>
<td>1.000</td>
<td>1.000</td>
<td>85420</td>
<td>1.000</td>
<td>1.000</td>
<td>39880</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>13360</td>
<td>1.967</td>
<td>0.984</td>
<td>43490</td>
<td>1.964</td>
<td>0.982</td>
<td>22650</td>
<td>1.761</td>
<td>0.880</td>
</tr>
<tr>
<td>3</td>
<td>8940</td>
<td>2.940</td>
<td>0.980</td>
<td>29660</td>
<td>2.880</td>
<td>0.960</td>
<td>17750</td>
<td>2.247</td>
<td>0.749</td>
</tr>
<tr>
<td>4</td>
<td>6680</td>
<td>3.934</td>
<td>0.984</td>
<td>23230</td>
<td>3.677</td>
<td>0.919</td>
<td>16020</td>
<td>2.489</td>
<td>0.622</td>
</tr>
<tr>
<td>5</td>
<td>5390</td>
<td>4.876</td>
<td>0.975</td>
<td>19620</td>
<td>4.354</td>
<td>0.871</td>
<td>15510</td>
<td>2.571</td>
<td>0.514</td>
</tr>
<tr>
<td>6</td>
<td>4500</td>
<td>5.840</td>
<td>0.973</td>
<td>17600</td>
<td>4.853</td>
<td>0.809</td>
<td>15650</td>
<td>2.548</td>
<td>0.425</td>
</tr>
<tr>
<td>7</td>
<td>3880</td>
<td>6.773</td>
<td>0.968</td>
<td>16400</td>
<td>5.209</td>
<td>0.744</td>
<td>16120</td>
<td>2.474</td>
<td>0.353</td>
</tr>
<tr>
<td>8</td>
<td>3420</td>
<td>7.684</td>
<td>0.961</td>
<td>15860</td>
<td>5.386</td>
<td>0.673</td>
<td>16690</td>
<td>2.389</td>
<td>0.299</td>
</tr>
<tr>
<td>9</td>
<td>2990</td>
<td>8.789</td>
<td>0.977</td>
<td>15590</td>
<td>5.479</td>
<td>0.609</td>
<td>17340</td>
<td>2.300</td>
<td>0.256</td>
</tr>
</tbody>
</table>

Table 5.15: Experimental Results for different modules of the Parallel Model PFC-DM2. [RuleBase = 350 rules].
<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Time(IF)</th>
<th>Speedup (IF)</th>
<th>Efficiency (IF)</th>
<th>Time(ILR)</th>
<th>Speedup (ILR)</th>
<th>Efficiency (ILR)</th>
<th>Time(FC)</th>
<th>Speedup (FC)</th>
<th>Efficiency (FC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35340</td>
<td>1.000</td>
<td>1.000</td>
<td>140230</td>
<td>1.000</td>
<td>1.000</td>
<td>51230</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>18000</td>
<td>1.963</td>
<td>0.982</td>
<td>73640</td>
<td>1.904</td>
<td>0.952</td>
<td>28860</td>
<td>1.775</td>
<td>0.888</td>
</tr>
<tr>
<td>3</td>
<td>11940</td>
<td>2.960</td>
<td>0.987</td>
<td>48420</td>
<td>2.896</td>
<td>0.965</td>
<td>21320</td>
<td>2.403</td>
<td>0.801</td>
</tr>
<tr>
<td>4</td>
<td>9030</td>
<td>3.914</td>
<td>0.978</td>
<td>37190</td>
<td>3.771</td>
<td>0.943</td>
<td>18270</td>
<td>2.804</td>
<td>0.701</td>
</tr>
<tr>
<td>5</td>
<td>7220</td>
<td>4.895</td>
<td>0.979</td>
<td>31800</td>
<td>4.410</td>
<td>0.882</td>
<td>17550</td>
<td>2.919</td>
<td>0.584</td>
</tr>
<tr>
<td>6</td>
<td>6025</td>
<td>5.866</td>
<td>0.978</td>
<td>27920</td>
<td>5.023</td>
<td>0.837</td>
<td>17820</td>
<td>2.875</td>
<td>0.479</td>
</tr>
<tr>
<td>7</td>
<td>5200</td>
<td>6.796</td>
<td>0.971</td>
<td>25710</td>
<td>5.454</td>
<td>0.779</td>
<td>18060</td>
<td>2.837</td>
<td>0.405</td>
</tr>
<tr>
<td>8</td>
<td>4520</td>
<td>7.819</td>
<td>0.977</td>
<td>24690</td>
<td>5.680</td>
<td>0.710</td>
<td>18470</td>
<td>2.774</td>
<td>0.347</td>
</tr>
<tr>
<td>9</td>
<td>4020</td>
<td>8.791</td>
<td>0.977</td>
<td>24530</td>
<td>5.717</td>
<td>0.635</td>
<td>18860</td>
<td>2.716</td>
<td>0.302</td>
</tr>
</tbody>
</table>

Table 5.16: Experimental Results for different modules of the Parallel Model PFC-DM2. [RuleBase = 450 rules].
Figure 5.43: The speedup curve for the IF module in the parallel model PFC-DM2.

[Rulebase = 350 rules].

Figure 5.44: The speedup curve for the IF module in the parallel model PFC-DM2.

[Rulebase = 450 rules].
Figure 5.45: The speedup curve for the ILR module in the parallel model PFC-DM2. 
*[Rulebase = 350 rules]*.

Figure 5.46: The speedup curve for the ILR module in the parallel model PFC-DM2. 
*[Rulebase = 450 rules]*.
CHAPTER VI

The Parallel Backward Chaining Models: Design and Implementation
6.1 Introduction

In this chapter we shall discuss the design and implementation of two parallel Backward Chaining models. These are:

1. The PBC-D1 Model: Parallel Backward Chaining Model with Dynamic Scheduling using technique 1.
2. The PBC-D2 Model: Parallel Backward Chaining Model with Dynamic Scheduling using technique 2.

The two techniques (technique 1 and 2) used to implement the dynamic scheduling strategy are described later in Sections 6.3 and 6.4.

The backward chaining models are suitable for a rule-based expert system, and they are domain independent. To support the use of these models, a "rule base compiler" has been built to translate a rule base in text format into the data structure needed by the system.

6.2 The Backward Chaining Process

One of the reasoning mechanisms used by rule-based expert systems is that of backward chaining. Backward chaining or 'goal-directed' inference starts with the desired goal and attempts to find evidence for this to be the case. Backward chaining is typically used in situations where the quantity of data is potentially very large, and where some specific characteristic of the system under consideration is of interest. Most typical are various problems of diagnosis, such as medical diagnosis or fault finding in electrical or mechanical equipment.

Consider the following example:
Example 1:

Assume that the knowledge base is as follows:

**RULEBASE:**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF A and B and C</td>
<td>THEN D</td>
</tr>
<tr>
<td>2</td>
<td>IF D and F</td>
<td>THEN G</td>
</tr>
<tr>
<td>3</td>
<td>IF L and M</td>
<td>THEN K</td>
</tr>
<tr>
<td>4</td>
<td>IF F</td>
<td>THEN B</td>
</tr>
<tr>
<td>5</td>
<td>IF B</td>
<td>THEN C</td>
</tr>
<tr>
<td>6</td>
<td>IF G</td>
<td>THEN H</td>
</tr>
<tr>
<td>7</td>
<td>IF I</td>
<td>THEN J</td>
</tr>
<tr>
<td>8</td>
<td>IF A and F</td>
<td>THEN H</td>
</tr>
</tbody>
</table>

**DATABASE (facts):**

- A is TRUE
- F is TRUE

Suppose the goal is to prove H i.e. prove that H is true.

On a sequential machine, a backward chaining system will try first to find a rule which proves H i.e. a rule which has H on its right-hand side, (as a conclusion).

The only candidates rules are 6 and 8, but 6 is encountered first. At this point we establish a new sub-goal of proving that G is true. If we can do this then it would follow that H were true by 'modus ponens'.

Our next sub-goals will be to prove that D and F are true (D and F → G).

Since F is true (fact), so it is only necessary to prove D. The whole proof proceeds as follows:

1. Trying to prove H.
2. Try rule 6 (IF G THEN H).
3. Trying to prove G
4. Try rule 2 (IF D and F THEN G).

5. F is true (F ∈ DATABASE), trying to prove D.

6. Try rule 1 (IF A and B and C THEN D).

7. A is true (A ∈ DATABASE), trying to prove B.

8. Try rule 4 (IF F THEN B).

9. F is true, so B is true.


11. Trying to prove C.

12. From rule 5 (IF B THEN C) we conclude that C is true since B is true.

13. Apply rule 1, concludes that D is true.

14. Apply rule 2, concludes that G is true.

15. Apply rule 6, concludes that H is true.

Note that we could prove H in one step from rule 8 (Since A and F are true). The point is that rule 8 was not first encountered and the system could not know in advance that it was going to be quicker to explore that rule than rule 6. Figure 1.1 illustrates the proof strategy, while Figure 6.2 illustrates the whole tree for the backward chaining process.

On a parallel machine, rule 6 and 8 could have been tried at the same time and hence, less time would be requested to prove H. This type of parallelism is known as OR-parallelism (see Chapter 4).

So backward chaining can be represented as searching through a branching network or tree. Trees may be searched in a number of ways. These are conveniently divided into a 'blind search' and 'informed search'. The latter is often called
Figure 6.1: Illustration of the proof strategy for (H is true) [Example 1]

Figure 6.2: The whole backward chaining mechanism tree [Example 1]
heuristic search. The two basic methods of blind search are called 'depth-first' and 'breadth-first' search (see Chapter 4). An AND/OR tree is a useful device for representing the behaviour of a rule-based expert system that works by problem decomposition, i.e. by decomposing high-level goals into a series of subgoals, each of which may have their own associated subgoals, and so on. Thus, to achieve high-levels goals, these subgoals should be achieved first. There are two types of nodes in an AND/OR tree: OR-nodes and AND-nodes. An OR-node is satisfied if just one of its associated subgoals (child-nodes) is satisfied, whereas, in the case of an AND-node, all the associated subgoals have to be satisfied in order for the node to be satisfied.

Finally, OR-parallelism is defined as the simultaneous exploration of multiple parts of a conjunctive goal. In Example 1, exploring rule 6 and rule 8 simultaneously to prove H is OR-parallelism, while exploring A, B and C simultaneously in rule 1 to prove D is AND-parallelism.

6.3 The Parallel Strategy

In Example 1, to prove the goal H, a backward chaining system, running on a sequential machine, will try rule 6 first and if this path will not prove the goal, it goes on to try rule 8 using a backtracking mechanism. With a multiprocessor system, we could start all the possibilities at once, by making one (or more) processors to consider rule 6(first choice), while the other processor(s) process rule 8. These OR choices, since they are independent, do not consume values from each other, so they can run with a minimum of synchronization.

As explained previously, a search problem can be represented by a AND/OR tree. The root of the tree represents the initial problem to be solved (the goal). the
non-terminal nodes are either AND nodes or OR nodes. An AND node represents a problem or subproblem that is solved only when all its children have been solved. An OR node represents a problem or subproblem that is solved when any of its children has been solved. In Figure 6.2 we have F and A are two terminal nodes, H is an OR node with two choices the first is rule 6 (node G) and the second is rule 8 (nodes A and F).

A typical shared memory multiprocessor has a moderate number of processors (Sequent Balance, for example). We want to utilize this class of multiprocessor systems in such way that backward chaining systems will run faster than on a single processor machine.

Among all the different types of parallelism, OR-parallelism seems to be the most promising for our problem and seems to offer a good potential for large scale and large-granularity parallelism. Moreover, parallelism in OR-parallel systems can easily be adjusted to be 'coarse-grained' and thus suitable for the current multiprocessors.

In this chapter, we propose a parallel inference mechanism based on a parallel backward chaining technique with OR-parallelism. The method used is based on a parallel search of the goal tree. The idea is to divide the search tree into subtrees, where each subtree corresponds to an OR node (choice node). All the OR-nodes generated during the parallel search are maintained in a task-queue with each OR-node representing one task. The processors share the task-queue, which is resident in the shared memory of the system. Each processor collects a task from the task-queue, process the task (This may generate new OR-nodes, which are then added
to the task-queue), and becomes idle again and looks for more work. The processor which becomes idle will check the task-queue for a task to perform, and if no task is available the processor has to wait for some tasks to appear in the task-queue.

Our aim is to make the model suited to 'coarse-grain' parallelism i.e. the subtree (task) assigned to each processor has to be sufficiently large to allow the benefits of parallelism to outweigh the overheads in distributing and collecting the tasks. Thus, allowing every goal to be OR-parallel, i.e. to add its OR-nodes (Choice nodes) to the task-queue, will make the parallelism too 'fine-grained'.

The strategy followed here is to allow the processor to split up its search subtree and generate new OR-nodes, i.e. new tasks, only if there is at least one processor in a waiting state, i.e. waiting for some tasks to appear in the task-queue. Thus, before a processor generates new OR-nodes, it checks the other processors status, and if it finds one or more processors waiting, it then generates the OR nodes and add new tasks to the task-queue.

This method has the following advantages:

1. Very little communication between the processors.
2. Better usage of the shared memory architecture.
3. A minimum synchronization overheads.
4. A dynamic distribution of tasks among the processors.
5. Reasonably large size for the tasks.

This method is well explained in the next two sections together with the design and implementation issues.
6.4 The Parallel Models: Design and Implementation

6.4.1 THE PBC-D1 MODEL

6.4.1.1 The Data Structures Used

The main data structures used are:

1. The Rule Base (RB).
2. The Data Base (DB).
3. The OR-Node-Set (ONS).
4. The Tree-Of-Nodes (TON).
5. The Processors-Status-Array (PSA).

1. The Rulebase: This is an array of records, where each record represents one rule. Each rule is a record with 3 attributes: the number of premises in the rule (Nprem), the list of premises and the list of conclusions. The rule structure used is the same as described before in Chapter 5 (Section 5.4.1), with each premise record having the additional field flag as explained in Section 5.4.2.

As an example consider the rule r₁:

\[ \text{IF } C_1 \& C_2 \& C_3 \text{ THEN A} \]

The internal representation of rule r₁ is shown in Figure 6.3.

2. The Database: This is an array of records called “Object-record”. Each record represents a fact and contains information about the object such as: name, value, and some other useful information.

3. The OR-Node-Set: This is an array of records called “OrNode-Rec”. Each record has the following attributes:
Figure 6.3: Internal representation of rule $r_1$

- **rule$n$b:** This is the number of the rule in hand (its index in RB).

- **node:** This is a record structure. It has two attributes, the object and the value given to this object.

- **generate-id:** This is the identification number (ID) of the process which generates the node.

- **processing-id:** This is the ID of the process which is processing the node.

- **solved:** This variable is used to mark the status of the node. It can take 3 values:
  
  -1: node is not solved.

  1: node is solved.

  0: not known yet (node is not processed yet).

- **level:** This is the level of the node in the node tree i.e. the level in the tree where the node is found (see Figure 6.4).
4. The Tree-Of-Nodes (TON): This is represented as an array with \( m \) elements called ‘Nodes-Tree’, where \( m \) is the number of processors on-line. Each element, a ‘Nodes-Tree’, is an array of records called ‘Node-Rec’. A ‘Node-Rec’ has the following attributes:

- \( \text{rulenb} \): This is the number of rules in hand (its index in RB).
- \( \text{node} \): This is a record with two attributes: the object and its value.
- \( \text{solved} \): This is to indicate whether the node is solved or not. It can have 3 values:
  -1: The node is not solved.
  1: The node is solved.
  0: It is not known, node not processed yet.
- \( \text{parent} \): The index of the node’s parent in ‘Nodes-Tree’.
- \( \text{choice-list} \): This is a linear list of records. Each record represents a choice node and has the following attributes:
- `rulenb`: The number of the rule which has the choice node in its conclusion part.
- `solved`: indicates whether the choice node is solved (1), not solved (-1), or not processed yet (0).
- `parent`: This is the index of the choice node's parent in Nodes-Tree.
- `first, last`: The indexes in Nodes-Tree of the first and the last nodes that derive from the choice node, i.e. are in the premise part of the rule numbered ‘rulenb’(RB[rulenb]).
- `parent-pos`: This is an Integer which indicates whether the parent of the node in hand is not in the choice-list (parent-pos= 0) or the parent-node’s position is in choice-list (parent-pos = n/n ≠ 0 ⇒ parent is the nth element in choice-list).
- `first, last`: These are the indexes in Nodes-Tree of the first and the last nodes that derive from the node in hand, i.e. they are in the premise part of the rule numbered ‘rulenb’. (Note that ‘rulenb’ for the node is not the same ‘rulenb’ used in choice-list for the choice-nodes).
- `result`: This is the result of investigating the OR node in hand i.e. the node and its associated choice-nodes in choice-list. The node is satisfied if it is solved or if any of its associated choice-nodes in its choice-list is solved. ‘result’ can have 3 values:
  - -1: node is not satisfied.
  - 1: node is satisfied.
  - 0: node not investigated yet.
• *level*: This is the level of the node in Nodes-Tree, i.e. its level in the AND/OR tree illustration of Nodes-Tree.

Consider the following example:

**Example 2:**

**RULEBASE:**

rule 1: IF A and B THEN C  
rule 2: IF D and E THEN C  
rule 3: IF F THEN B  
rule 4: IF G and H THEN A  
rule 5: IF K and L THEN B

Suppose that processor 2 (*P*₂) is to investigate the goal 'C is true'. *P*₂ uses the structure Tree-Of-Node [2] to represent its search tree, i.e. the tree of the nodes (goals, subgoals) generated by decomposing the goal 'C is true' into subgoals, and then the subgoals into sub-subgoals and so on. Tree-Of-Node [2] is of the type 'Nodes-Tree' and is illustrated in Figure 6.5. This figure can be interpreted in the following way: Find all the rules with C in their conclusion part. There are 2 candidate rules (rule 1 and rule 2). Add rule 1 to Nodes-Tree [1] (*rulenb = 1*) and rule 2 as the first element of the choice-list of Nodes-Tree [1]. Thus, the *rulenb* field in the table Nodes-Tree [1] is filled with 1 and the *rulenb* field in the first element of choice-list is filled with 2. The choice-list is used to maintain all the choice-nodes (OR nodes) of one goal/subgoal. For the goal/subgoal to be satisfied (*result = 1*), it is enough to have the node (in Nodes-Tree) or one choice-node (in the choice-list)
satisfied (solved = 1 for the node or for any of the nodes in the choice-list). The first row in the table (Nodes-Tree [1]) will be interpreted as follows:

- **rulenb** = 1: The number of the first located rule which has the node (the goal C) in its conclusion part.
- **node** = C: The node in hand (goal/subgoal) is equal to C.
- **solved** = 0: It is not known yet whether the node is solved or not.
- **result** = 0: We do not know if the goal (the node C) is satisfied or not. A node is satisfied (result = 1) if the node is solved (solved = 1) or any of its choice-nodes is solved.
- **parent** = 0: The node C does not have a parent-node in Nodes-Tree, thus the field parent is initialized to zero.
- **parent-pos** = 0: In the same way parent-pos is initialized to zero.
- **first** = 2: This indicates the address in ‘Nodes-Tree’ of the first child of C.
- **last** = 3: This indicates the address in ‘Nodes-Tree’ of the last child of C.
- **level** = 0: The node C is found at level zero (see Figure 6.5).

**choice-list**: This list consists of one choice-node which has:

- The rulenb equal to 2, i.e. the number of the ‘second located’ rule with C in its right-hand side (rule 2).
- The node is not investigated yet: solved = 0.
- The parent-node’s address is initialized to zero, since the node C does not have a parent in Nodes-Tree.
- The address in Nodes-Tree of its first child is equal to 4.
- The address of its last child is equal to 5.
On the other hand node B in row 3 has its parent-node situated in the first row of Nodes-Tree. To determine the position of this node i.e. whether it is in the choice-list or not, we should read the field parent-pos. This is equal to zero, which indicates that the parent-node is not in the choice-list. Thus, the parent-node of B is the node C in rule1 (rulenb = 1), situated at level zero, with first child address is equal to 2 and last child address is equal to 3.

On the contrary, node E in row 5 has its parent-node situated in the first row of Tree Node and in the first element of choice-list (parent= 1, parent-pos = 1); thus its parent-node is the node C in rule 2 (rulenb = 2), with first equal to 4 and last equal to 5. In addition, both nodes B and E are found on level 1 of the search tree (level = 1). See Figure 6.5.

Before finishing with this section, we shall give the following definitions:

Definition 1:

Let A be a node, and r1, r2 and r3 three rules which have A in their conclusions. We refer to node A in r1 as a couple (A, 1) to indicate that this is the node found in rule r1. In the same way, the other two nodes are (A, 2) and (A, 3). Suppose that we were to find all the rules in RB with A in their conclusions, and that the node (A, 1) was found first. We call the node (A, 1) the ‘original’ node and nodes (A, 2) and (A, 3) are called ‘choice-nodes’. The ‘original node’ is added to Nodes-Tree, while the ‘choice-nodes’ are added to the choice-list of the same element of Nodes-Tree. The node (A, 2) is kept in the first element of choice-list and node (A, 3) is kept in the second. In Example 2, the node (C, 1) is added to Nodes-Tree [1], while node (C, 2) is added to the choice-list of Nodes-Tree [1].
Note 1:

A new element is added to the end of the choice-list (not in the front).

Definition 2:

Let \( A \) be a node. The parent of node \( A \) is defined as a triplet: (node, parent, parent-pos), where 'node' is the content of the parent-node (object, value, ...), 'parent' is the address in Nodes-Tree of the parent-node, and 'parent-pos' is the position or location of the parent-node in Nodes-Tree [parent]:

- \( \text{parent-pos} = 0 \Rightarrow \text{parent-node is in Nodes-Tree [parent] but not in the choice-list.} \)
- \( \text{parent-pos} = n/n > 0 \Rightarrow \text{parent node is in the } n^{th} \text{ element of the choice-list of Nodes-Tree [parent].} \)

In Figure 6.5, the parent-node for node D in row 4 is defined as the triplet (C, 1, 1). This is node C in row 1 (Nodes-Tree [1]) and in the first element of choice-list (rulenb = 2, first = 4, last = 5).

Definition 3:

Let \( A \) and \( B \) be two nodes; the node \( B \) is a 'child-node' of \( A \), if there is a rule \( r_i \) in RB, that has \( B \) in its premises list and \( A \) as its conclusion, i.e., \( r_i \) has the form:

\[ \text{IF } B \& \ldots \text{ THEN } A. \]

In Example 2, \( D \) is a child-node of \( C \). \( E \) is another child-node of \( C \).

B is a 'child-node' of \( A \) \( \iff \exists \ r_i \in RB / B \in \text{premise} (r_i) \text{ and } A \in \text{conclusion} (r_i). \)

Corollary 1:

Let \( r_i \) be a rule. \( r_i / r_i: \text{IF } C_1 \& C_2 \& C_3 \text{ THEN } A. \)

The 'child-nodes' of \( A \) are \( C_1, C_2, \) and \( C_3 \). In other words, the child-nodes are the nodes in the premise of the rule.
5. The Processors-Status-Array (PSA):

This is represented as an array of Boolean values. It consists of \( m \) elements, where \( m \) is the number of processors on-line. To find out about the status of processor \( p \), we read the contents of \( \text{PSA}[p] \). There are two possibilities:

1. \( \text{PSA}[p] = \text{TRUE} \rightarrow \text{Processor } p \text{ is waiting.} \)
2. \( \text{PSA}[p] = \text{FALSE} \rightarrow \text{Processor } p \text{ is active (not waiting).} \)

6.1 The Model and its Implementation

We assume that:

- The rules are numbered from 1 to \( n \).
- The processors are numbered from 1 to \( m \).
- The facts needed by the system during its reasoning process are collected by the "User-Interface" (UI) module at the beginning of the consultation. This is done for performance measurement reasons.

The PBC-D1 model consists of 6 main modules (see Figure 6.6). These are:

1. The User-Interface (UI) module.
2. The Initialize-Flags (IF) module.
3. The Backward-Chaining (BC) module.
4. The Parallel-Search (PS) module.
5. The Get-Results (GR) module.
6. The Show-Results-Explain (SRE) module.
Figure 6.5: Internal representation of the AND/OR tree [Example 2]
1. The User-Interface (UI) module

The main job of this module is to communicate with the user (or any other input sources) to collect the facts needed by the system throughout the consultation.

The collected facts are kept in the database which resides in the shared memory of the system. This module is activated only at the beginning of the system run-time, and is kept separated from the backward-chaining inferencing cycle, for performance measurement reasons.
2. The Initialize-Flags (IF) module

This initializes the flags of the rules premises. It matches the rules premises against the initial facts in DB, and then set the premises to either TRUE or FALSE according to the result of the match test. The algorithm for this module is:

![Diagram of step 1: Send a rule to the processor](a) Send a rule to the processor
(b) Process the rule

$P_i$: Processor $i$

**Figure 6.7: Partitioning the rules in the IF module**

**Step 1** Define the number of processors to be used to execute the job in parallel. Assume that this number is $q$. ($1 \leq q \leq m$).

**Step 2** Set up the $q$ processors for work. (Call `m-set-nprocs`).

**Step 3** Create $q$ processes to execute the job. (Call `m-fork`).

**Step 4** Execute the job (Initialize the flags) in parallel. This is done using data-partitioning with the static scheduling technique. The algorithm for this step is (see Figure 6.7):
1. Call ‘m-get-numproc’ to determine the number of processes created by the system to carry out the job. (This number is q).

Cycle 0

2. FOR each process DO

   a) Call ‘m-get-myid’ to find out the process’s ID (Call this number N).

   b) Determine the process task, i.e. figure out which rule to process. The rule number (I) is determined as follows:

   \[ I = N + \lambda q \mod q \leq \lambda \text{ and } N + \lambda q \leq n. \]

   Task = Process (rule_i).

   \( \lambda \) is/at cycle 0, \( \lambda = 0 \); at cycle 1, \( \lambda = 1 \); and so on.

   c) IF I > n THEN no work is done and the process becomes idle.

   ELSE

   • Assign rule_i to the process with ID = N

   • Process rule_i:

     • For each condition \( C_t \) in the premise part of rule_i DO

     • Match \( C_t \) against the facts in DB.

     • IF \( C_t \) is satisfied THEN set the flag of \( C_t \) to TRUE, otherwise set the flag to FALSE.

     • Wait for the other \((q - 1)\) processes to finish their tasks.

3. All q processes have finished their tasks.

4. IF there are still more rules to be processed THEN

   • Start the next cycle.

   • Repeat 2 to 4 for the \((n - q)\) unprocessed rules.
3. The Backward-Chaining (BC) module

This organizes and controls the overall reasoning process. This is the Manager module which supervises the last three modules (PS, GR and SRE) and coordinates their activities. The BC module is the one to decide when to stop the consultation and give results. It cycles through the following phases (see Figure 6.8):

Phase 1:

Scan the rulebase RB to find all the rules that have the goal in their right-hand side (conclusion). The candidate rules are then maintained in an array called ‘rule-set’. If the array rule-set is empty, then no candidate rule is found and consequently the goal is not valid and a failure message is sent. Otherwise, start Phase 2.

Phase 2:

The algorithm for this phase is:

Step 1 Check if the goal exist in DB, and if so, the goal is then achieved. If the goal does not exist in DB, then start Step 2.

Step 2

1. Define the number of processors to be used to execute the job in parallel.
   Assume that this number is $q \ (1 \leq q \leq m)$

2. Set up the $q$ processors for work. (Call m-set-nprocs).

3. Initialize all the elements of PSA to TRUE. This indicates that all the processors are waiting (not active).

   \[
   \{\text{FOR } I = 1 \text{ To } q \text{ DO} \}
   \]

   \[
   PSA \ [I] \leftarrow \text{TRUE}
   \]

4. Add the content of ‘rule-set’ to ONS, i.e. add the goal node and its ‘choice-nodes’ (OR nodes) to ONS.
Scan the rulebase to find the rules that have the goal in their right-hand sides. Store the rules in 'rule-set'.

- If the 'rule-set' is empty, STOP and send a failure message.
- If the goal is in the DB, yes, goal is achieved (Trivial case).
- Define the number of processors to process the job.
- Set up the processors for work.
- Initialize PSA elements to TRUE. (processors are waiting)
- Add the nodes corresponding to the rules in 'rule-set' to ONS.
- Initialize the variables 'Halt' and 'SolFound' to FALSE.
- Activate the PS module. (Fig. 6.11)
- Terminate the processes.
- Get the result of the consultation i.e. Activate the GR module. (Fig. 6.17)
- Show results and explanations if needed (SRE module).

Figure 6.8: The Backward Chaining module in PBC-D1

5. Initialize the variables 'Halt' and 'SolFound' to FALSE.

6. Start the next module PS, which assigns the Backward Chaining process.
to the processors to execute it concurrently.

7. Broadcast the results of processing the OR nodes in ONS to the module GR, which then concludes the result of the consultation.

8. Start the module SRE, which shows the results and give the necessary explanations, if needed.

4. The Parallel Search (PS) module

This is the module which carries out the backward reasoning. PS partitions the job among the \( q \) processors, schedules their activities, and sorts out the synchronization and communication problems. The concept of this module is depicted in Figure 6.9. All the processes share one 'task-queue'. ONS is used to implement the concept of the task-queue. It contains the OR-nodes (choice nodes) to be processed in parallel. A task is defined as processing one OR-node from ONS. At start, ONS contains all the choice-nodes of the goal in hand (elements in rule-set).

Two indexes are used with ONS. These are:

a. ‘index-OR’ which contains the address of the next unprocessed node in ONS.
   This is the address of the node to be processed next.

b. ‘Numb-OR’ which is the address of the last node in ONS. This is also the total number of nodes in ONS.

The task-availability in the task-queue is determined as follow:

- \( \text{index-OR} > \text{Numb-OR} \rightarrow \text{there is no available tasks in the task queue.} \)
- \( \text{index-OR} \leq \text{Numb-OR} \rightarrow \text{there are some available tasks in the task-queue.} \)

Number of tasks = \( \text{Numb-OR} - \text{index-OR} + 1 \).
Definition 4:

A node is called ‘terminal’ node if it does not have any child-node. This means that there is no rule in RB, which has the node in its right-hand side. Thus:

```
Get the processor ID (call this 'proc-id')

WHILE (not Solfounq) & (not Hail) DO

Begin WHILE

Start a critical section
(Leck a LOCK)

Is there any available task in task-queue ??

No

The processor has to wait.
Status is WAIT

Yes

Are all the other processors Waiting ??

No

Job is done, halt the execution
Halt ← TRUE

Exit the critical section.
(unlock the LOCK)

Yes

Take the first task in the task queue

Change the processor status to ACTIVE

Exit the critical section.
(unlock the LOCK)

PROCESS the task (fig. 6.11)

Make another request

End WHILE

END
```
A node \( n \) is terminal
\[
\equiv \forall r_i \in RB; \; n \notin \text{conclusion of } r_i
\]
\[
\equiv \; n \; \text{has no child nodes.}
\]

**Definition 5:**

A node is said to be 'solved' if one of the following conditions holds:

1. The node is in the set of terminal nodes and exist in DB. (In this case the node has no 'child-nodes').
2. The node has all its 'child-nodes' (AND nodes) solved.
3. The node has 'choice-nodes' (OR nodes) and any one of these nodes is solved.

**Definition 6:**

A node is said to be 'not solved' if one of the following conditions is true:

1. The node is terminal and is not in DB.
2. The node has one or more of its 'child-nodes' not solved.
3. The node has choice-nodes and all these nodes are not solved.

**Definition 7:**

Let \( A \) be a node. Suppose that \( A \) has some choice-nodes. The 'OR-parallel' node corresponding to \( A \) is such that:

1. It has the node \( A \) and its choice-nodes as OR-nodes.
2. It is represented in TON as follows:
   a) The node \( A \) is in TON
   b) The choice-nodes of \( A \) are in the choice-list.
3. It is solved, if node \( A \) is solved or any one of its choice-nodes is solved. The field 'result' in the node record is used to mark if the OR-parallel node is solved or not.
The processors share the same ‘task-queue’ (this is ONS) which is resident in the shared memory of the system (see Figure 6.10). The processors status are determined by the content of the array PSA which also resides in the shared memory. Moreover, the processors share a global variable called ‘Halt’ which determines whether to halt the execution or not. At start, ‘Halt’ is initialized to FALSE.

![Figure 6.10: The concept of task partitioning in PS](image)

Each processor cycles through the following steps:

WHILE (Not Halt) and (Not SolFound) DO

{Begin WHILE}

Step 1 Start a critical section (lock a LOCK).

Step 2 Check the task-queue for a task to perform.

Step 3

IF no task is available THEN

1. Wait (change the status to WAIT). {PSA [processor] ← TRUE }
2. Check the status of the other processors.

3. IF they are all waiting THEN
   
   - Halt the execution (job is finished). [Halt ← TRUE],
   - Exit the critical section (unlock the LOCK).

4. ELSE
   
   - Make another task request.

Step 4 ELSE {task is available}

1. Collect the task (this is ONS [index-OR]).
2. Increment index-OR by one.
3. Change the processor status to Active {PSA [processor]← FALSE}.
4. Exit the critical section (unlock the LOCK).
5. Process the task. This action might generate new tasks.
   
   • IF new tasks are generated THEN
     
     - Start a critical section (lock a LOCK)
     
     - Add the new tasks to the end of the task-queue (add the OR-nodes to ONS, and update Numb-OR accordingly).
     
     - Exit the critical section (unlock the LOCK).

{End WHILE}

This algorithm is depicted in Figure 6.9.

Definition 8:

Let \( r_i \) be a rule. A node is corresponding to \( r_i \) and vice versa, if the node \( rulenb \) field is equal to \( i \).

A node \( N \) corresponds to \( r_i \rightarrow N.rulenb = i \).
(I) The Process Form

Each process, running on one processor, has the following form:

1. Local variables and data

These are the data and variables local to each processor. The main local variables used are:

(i) ‘proc-id’: which keeps the ID of the processor in hand.

(ii) ‘task-position’: which is the position of the task (OR-node) being processed, in the task queue, i.e. its index in ONS.

(iii) The set $S_I$: This is an array structure with $S_I[0]$ holding the number of elements in $S_I$. Each element in $S_I$ contains the number of a rule, i.e. the index of a rule in RB.

(iv) ‘L-index’, ‘L-NumbNode’: $L$-index is an index for TON [processor] structure. This is the address in TON [processor] of the current processed node. $L$-NumbNode holds the number of the nodes in TON [processor] at a certain time.

(v) The variables: rule, rule-I, match used before in PFC-S1 and PFC-SM1 in Chapter 5.

In addition to these variables and data, each processor contains the essential software to process the task in hand, to send and receive data from shared memory, and finally to synchronize with the other running processes.

2. Global shared variables and data

These are the variables and data resident in the shared memory of the system and accessed by all the processes. Each process uses the following global shared data:
(i) The Rulebase (RB), the Database (DB), the OR-Node-Set (ONS), and the Processors-Status-Array (PSA).

(ii) The Tree-Of-Nodes (TON): This is an array of \( m \) array structures, where \( m \) is the number of processors on-line. Each array in TON is used by one processor to process its tasks. Thus, processor number \( I \), uses TON\([I]\) to process its tasks, \((0 < I \leq m)\).

(iii) The variable SolFound: This indicates if the goal is achieved or not.

(iv) The variables Halt: This is used by the processes to determine whether the job is completed or not.

3. The process algorithm

The algorithm for each process is the following (see Figure 6.9):

1. Call the parallel programming Library (PPL) routine “m-get-myid” to determine the process ID. This value will be stored in the local variable proc-id.

2. WHILE (NOT SolFound) and (NOT Halt) DO

{Begin WHILE}

- Lock a LOCK.
- Check the task-queue for a task, i.e. compare index-OR with Numb-OR, to find out whether all the OR-nodes in ONS are processed or not.
- IF a task is available, i.e. (index-OR \( \leq \) Numb-OR) THEN
  - task-position = index-OR.
  - Collect the task ONS [task-position].

269
o Increment index-OR.
o Change the processor status to ACTIVE: PSA[proc-id] ← FALSE.
o Unlock the LOCK.
o Process the task. This step is explained in the next section (see Figure 6.11).

• ELSE {no task is available} i.e. (index-OR > Numb-OR)
o Change the processor status to WAIT: PSA [proc-id] ← TRUE.
o Check the status of the other processors.
o IF they are all waiting THEN
  • Halt the execution, job is completed. (Halt ← TRUE).
  • Unlock the LOCK.
o ELSE
  • Make another request.

{End WHILE}  

(II) Process The Task Phase

The algorithm for this phase is depicted in Figure 6.11. The task is the node ONS [task-position] (call it nodeT) and the processor ID is equal to proc-id. This phase consists of the following steps:

Step 0 Set the processing-id of the task in ONS to proc-id. (To mark that the (proc – id)th processor is processing the task).

Step 1 Add nodeT to the top of TON [proc-id].

Step 2 If nodeT exist in DB then it is solved. Thus mark it solved i.e. 
(result = 1 and solved = 1). Go to Step 15(8).
Step 3 If nodeT does not exist in DB then expand nodeT by generating its child-nodes. The child-nodes are the premises of the $I^{th}$ rule in RB, $I$ is such as: $I = \text{nodeT.rulenb}$. Add nodeT’s child-nodes to TON[proc-id] and update ‘L-Numb’ accordingly. Link the child-nodes to nodeT ‘the parent node’, i.e. fill in the ‘parent’ field in the child-nodes, with the index of nodeT in TON [proc-id]. The ‘parent-pos’ field is filled with zero (Parent in TON and not in choice-list).

- Set the fields ‘first’ and ‘last’ in nodeT to the addresses in TON [proc-id] of the first and the last child-node.

- Set the level field in the child-nodes to the parent-node level plus one.

\[
\text{level} \leftarrow \text{nodeT.level} + 1
\]

- Update L-Numb:

\[
\text{L-Numb} \leftarrow \text{L-Numb} + \text{number of child-nodes} = \text{L-Numb} + \text{RB[I].Nprem}
\]

Step 4 Start with the first non-processed node. (Call this node ‘nodeC’).

Step 5 IF nodeC exist in DB THEN the node is solved. Mark nodeC solved, and go to Step 13.

Step 6

IF nodeC does not exist in DB THEN:

- Scan RB to find all the rules that have nodeC in their right-hand side.

- Store the candidate rules in the set S1.
Start

Add the task (nodeT) to the Top of TON [proc-id].

Node ≠ DB?

Yes

Mark the node 'solved'.

No

Expand the node, i.e. generate its child nodes. Add the child nodes to TON [proc-id]. Update their levels.

WHILE there are still non-processed nodes in TON [proc-id] DO

Begin WHILE

Take the first non-processed node. (Call it nodeC)

nodeC ≠ DB?

Yes

Mark the node 'solved'.

No

Scan the rulebase to find the rules that have nodeC in their right-hand side. The found rules are kept in a set S1.

Is S1 empty ??

Yes

The node is a terminal node and not in DB thus it is not solved. Mark the node 'Unsolved'.

No

Investigate the rules in S1.

A satisfied rule is found ?

Yes

Set the node 'rulenb field' with the number of the rule in hand.

Mark the node solved.

No

A satisfied rule is not found.

Continue next page

Forget about the other rules in S1 i.e. the choice nodes corresponding to the rules in S1.

GoTo End WHILE

end WHILE

The node is a terminal node and not in DB thus it is not solved. Mark the node 'Unsolved'.

Skip all its AND nodes in TON [proc-id].

GoTo End WHILE

end WHILE

Continued on next page
Consider the first rule in $S_1$, set the node ruleNum field with the number of this rule.

Add the child nodes of the node to TON [proc-id]. Update their levels.

Is there any processor waiting??

Yes
Add the choice-nodes, i.e. nodes corresponding to the other rules in $S_1$ to ONS. Thus, add new tasks to the task-queue.

No
Add the choice nodes corresponding to the other rules in $S_1$, to the choice-list of the node in hand
Add their child nodes to TON [proc-id]. Update their levels.

End WHILE

Conclude Backward in TON [proc-id], from the child nodes to their parents as to transfer the results upward in the tree. Update the corresponding rules in RB accordingly. (Fig. 6.12)

Broadcast the result of processing the task node to the task-queue ONS.

Check ONS to see if the goal node, or any of its OR nodes, is solved. If so, set SolFound to TRUE.

Figure 6.11: Processing the task in PS module
Step 7

IF S1 is empty (S1[0] = 0) THEN

- The node ‘nodeC’ is a terminal node and not in DB, thus it is not solved. So mark nodeC unsolved. (result = -1, solved = -1).

- Skip all its associated AND-nodes, i.e. the other child-nodes. These are the child-nodes of the parent-node (nodeP) of nodeC. The algorithm for this step is:

\[
\begin{align*}
\text{nodeP} &\leftarrow \text{nodeC.parent} \\
\text{position} &\leftarrow \text{nodeC.parent-pos} \\
\text{IF position} = 0 \text{ THEN} \\
\text{L-index} &\leftarrow \text{TON[proc-id,nodeP].last} \\
\text{ELSE} \\
\text{Go To the (position)th element (Ep) in the choice-list of nodeP} \\
\text{L-index} &\leftarrow \text{Ep.last}
\end{align*}
\]

- Go To Step 13.

Step 8 The set S1 is not empty (S1[0] \neq 0). Investigate the rules in S1, i.e. check the flags in the premises of the rule, if they are all equal to true then the rule is satisfied. If any of these rules is satisfied (Assume the satisfied rule number is equal to I) Then:

- Set the ‘ruenb’ field in nodeC to I (nodeC.ruenb\leftarrow I).

- Mark nodeC ‘solved’, i.e. (nodeC.result = 1, nodeC.solved = 1).

- Forget about the other rules in S1. In other words, since the node is solved, there is no need to investigate the other choice-nodes of
nodeC (OR-nodes). Consequently, these choice-nodes are not added to nodeC.choice-list.

- Go to Step 13.

Step 9 None of the rules in S1 is satisfied.

Start with the first rule in S1, S1[1]:

- Set the rule nb field in nodeC with S1[1].
- Expand nodeC, generating its child-nodes. These are the nodes in the premise of rule S1[1].
- Add the child-nodes to TON [proc-id], and update ‘L-Num’ accordingly.

\[
{ I = S1[1] } \\
L-Num \leftarrow L-Num + RB[I].Nprem \]

- Link the child-nodes to its parent-node nodeC.

FOR each child-node DO

- Fill the parent field in the child-node with the address of nodeC in TON.
- Fill the ‘parent-pos’ field with zero (parent not in choice-list).
- Set the level field to the parent-node level plus one.

\[
level \leftarrow nodeC.level + 1. \]

Step 10 Check PSA to see if there is any processor waiting.

Step 11

IF (no processor is waiting, i.e. all processors are active) THEN:

- Add the choice-nodes of nodeC to the choice-list.
• FOR each choice-node \([\text{nodeC.node, } S1[\ell]]; 1 < \ell \leq S1[0]\) DO

  o Expand the node and generate its child-nodes.
  
  o Add the child-nodes to TON[proc-id]. Update 'L-Numb' accordingly.
  
  o Link the child-nodes to their parent-node. The algorithm is:

    FOR each child-node DO
    
      . Set 'parent' field to the address of the choice node in TON.
      
      This is nodeC address.
      
      . Set the 'parent-pos' field to the position of the choice-node in the choice-list. (parent-pos = 1 if the node is the first in the choice-list; parent-pos = 2 if the node is the second, and so on).
      
    o Update the level of the child-nodes. The algorithm is:

    For each child-node DO
    
      . Set the field 'level' to the level of the choice-node plus one.

      \[\text{child-node.level} \leftarrow \text{choice-node.level} + 1\]

Step 12

IF (one or more processors are waiting) THEN:

• Lock a LOCK.

• Select the choice-nodes that do not exist in ONS. In other words, remove from S1 all the nodes that are already in ONS. The reason for
this, is to prevent the processors from processing the same 'branch' of the search tree (the same OR-node) more than one time. To understand this point see Example 3.

- Add the choice-nodes of nodeC to ONS. These nodes are of the following form: choice-node \( \equiv (\text{nodeC.node, } S_1[\ell]) \) with \( 1 < \ell \leq S_1[0] \). In other words, we have added new tasks to the task-queue.

- Update \( \text{Numb-OR} \). \( \{ \text{Numb-OR} \leftarrow \text{Numb-OR} + (S_1[0] - 1) \} \).

- Set the field 'generate-id' to 'proc-id' for all the new added nodes.

- Unlock the LOCK.

**Example 3:**

Let \( r_i, r_j \) and \( r_k \) be 3 different rules in RB. These rules are such as:

\[
\begin{align*}
  r_i : & \quad \text{IF} \quad C_1 \& C_2 \& C_3 \quad \text{THEN} \quad A_1 \\
  r_j : & \quad \text{IF} \quad C_4 \& C_2 \& C_5 \quad \text{THEN} \quad A_2 \\
  r_k : & \quad \text{IF} \quad C_6 \& C_7 \quad \text{THEN} \quad C_2
\end{align*}
\]

The search trees for the nodes \( A_1 \) and \( A_2 \) are depicted in Figures 6.12a and 6.12b respectively.

Now suppose that we have two processors, \( P_1 \) and \( P_2 \) such as: \( P_1 \) is processing \( A_1 \) and \( P_2 \) is processing \( A_2 \). IF \( P_1 \) has already generated \( C_2 \) to ONS, i.e. add the node \((C_2, r_k)\) to ONS, then we shall make sure that \( P_2 \) is not going to add the same node to ONS. This is done, by checking first the content of ONS to verify that \((C_2, r_k)\) is not present and only then the node will be added to ONS. In our example, \( P_2 \) finds that \((C_2, r_k)\) is already present in ONS, and thus it removes it from its set \( S_1 \), and consequently, \( P_2 \) does not add the node to ONS.
Step 13 Increment 'L-index' by one. \{ L-index \leftarrow L-index +1 \}.

Step 14 IF there are still non-processed nodes in TON[proc-id] THEN goto Step 4. To determine if there are non-processed nodes in TON \[proc-id\] we compare \( L-index \) with \( L-Numb \), and subsequently:

- \( L-index > L-Numb \) \rightarrow All the nodes in TON \[proc-id\] are processed.
- \( L-index \leq L-Numb \) \rightarrow there are non-processed nodes in TON \[proc-id\].

Step 15 Drive backward in TON[proc-id] from the child nodes to their parent-nodes as to transfer the results from the child-nodes to their parent-nodes in TON \[proc-id\] (see Figure 6.13). Update the flags in the corresponding rules accordingly. As a result we will conclude the result of processing the node ‘nodeT’. The algorithm for this step is:

1. Start with the last node in TON \[proc-id\].

\[ \text{position} \leftarrow L-Numb. \]
2. Go To the parent of the node:

- \( \text{address-P} \leftarrow \text{TON}[	ext{proc-id,position}].\text{parent}. \)
  
  \{ \text{address-P is the address of the parent-node in TON}. \}

- \( \text{node-P} \leftarrow \text{TON} [\text{proc-id, address-P}] \) \{ \text{node-P is the parent-node}. \}

3. Transfer the solvability of the node to the corresponding rule in RB.

a. Get the number (rule\(_f\)) of the rule corresponding to the parent-node.

\[ \text{rule}_f \leftarrow \text{node-P}. \text{rule}_b \]

b. Go To the rule with index equal to rule\(_f\) in RB.

\[ \text{rule} \leftarrow \text{RB}[\text{rule}_f] \]

\[ \text{Premise} \leftarrow \text{rule.prem} \] \{ get the premise of the rule \}.

c. Update the flags in the premises of the rule according to the solvability of the child-nodes of node-P. The algorithm for this step is:

- \( \text{counter} \leftarrow 1 \).

- \( \text{FOR} \ I = \text{node-P}. \text{first} \ \text{To} \ \text{node-P}. \text{last} \ \text{DO} \) \{ the child-nodes \}

\{ \text{Begin FOR} \}

- \( \text{child} \leftarrow \text{TON} [\text{proc-id, I}] \) \{ child is a child-node \}.

- \( \text{IF} \ \text{child.result} = 1 \ \text{THEN} \) \{ child-node is solved \}

  - Set the flag in the \( (\text{counter})^{th} \) condition to TRUE.

  \[ \text{Premise}[\text{counter}].\text{flag} \leftarrow \text{TRUE} \]

  \[ \text{Note that the first condition correspond to the first child, the second condition to the second child, and so on.} \]

- \( \text{ELSE} \) \{ child is not solved \}

279
Check first if the current condition's flag is not set to 'true' by another processor, and if so leave it unchanged whatever the result of the child-node is:

- IF Premise [counter].flag <> TRUE THEN

  Premise [counter].flag ← FALSE {set the flag to FALSE}

  o counter ← counter +1

{End FOR}

4. Check the solvability of the OR-parallel node corresponding to node-P:

a. Check if all the child-nodes are solved (result = 1). If so, mark the parent-node, and the corresponding OR-parallel node as solved.

  node-P.result ← 1; node-P.solved ← 1.

b. DO the same for the choice-nodes of node-P i.e. nodes in choice-list of node-P, providing that this list is not empty. The algorithm is:

- IF choice-list is not empty (node-P.choice-list ≠ nil) THEN

  o FOR every choice-node (C-node) in the list DO

    - Get the number (rule_i) of the rule corresponding to C-node.

      rule_i ← C-node.rulenb

    - Update the flags in the premises of the rule according to the results of the child-nodes of C-node i.e. the nodes between 'C-node.first' and 'C-node.last' (see (3.c)).

    - Check if all the child-nodes of C-Node are solved, and if so, mark C-node solved, and set the field 'result' in node-P to one. Note that the result field in node-P is set to one if the
node or any of its choice-nodes is solved (OR-parallel node corresponding to node-P).

\[ \text{C-node.solved} \leftarrow 1; \text{node-P.result} \leftarrow 1. \]

5. IF neither node-P nor any of its choice-node is solved, THEN

- Mark the OR-parallel node as not solved (node-P.result = -1).

6. Move upward in TON [proc-id] to the node which comes just before the first child of node-P.

\[ \text{position} \leftarrow \text{node-P.first} - 1 \]

7. IF it is not the first node in TON[proc-id], THEN

Do the same operation (2 to 6) for the new node TON[proc-id, position]:

\[ \text{IF position} > 1 \text{ THEN Go To 2. ELSE Go To the next step (8).} \]

8. Get the result of processing the task (nodeT).

\[ \text{processing-result} \leftarrow \text{TON [proc-id, 1].result} \]

9. Broadcast the processing result to the task-queue.

\[ \text{ONS [task-position].result} \leftarrow \text{processing-result} \]

Step 16 Check if the goal G is achieved. If so, set SolFound to TRUE:

- Check ONS to see if the goal is achieved, i.e. the goal node or any of its OR-nodes is solved.

- IF (goal is achieved) THEN

\[ \text{SolFound} \leftarrow \text{TRUE} \]
Start with the last node in TON [proc-id]

GOTO the parent of the current node (call the parent node node-P)
goto the rule which corresponds to node-P. (q \in \mathbb{R}B \land \text{node-P.rules})

Update the flags in the premises of the rule according to the solvability of the child nodes of node-P.

Pass the results from the child nodes to their parent. i.e. Conclude the solvability of the parent node.

Does node-P have any choice nodes, i.e. choice list is not empty??

For every choice node C-node in choice-list DO

Goto the rule that corresponds to C-node. (q \in \mathbb{R}B \land \text{C-node.rules})

Update the flags in the premises of the rule according to the solvability of the child nodes of C-node.

Pass the results from the child nodes to their parent C-node, i.e. Conclude the solvability of C-node.

IF neither node-P is solved nor any of its choice nodes is solved, then mark the OR-parallel node as not solved.

Move upward in TON [proc-id] to the node which comes just before the first child of node-P (node-P.first + 1).

Did we reach the Top of TON [proc-id]??

Conclude the result of the node on top of TON [proc-id] (the task node)

END

Figure 6.13: Concluding the results of processing the task-node

282
(III) Overview of The PS Module

The algorithm used in PS can be understood as follows (see Figure 6.14): starting with a subgoal/goal, the task node, we first expand the node, always driving backward (downward in the tree) from the node to its child-nodes, and then from the child-nodes to their children and so on until no further expansion is possible. This operation, will generate the search tree for the task node, which has the structure of an AND/OR tree. This tree is represented with the structure TON[Pi], where Pi is the ID of the processor which is processing the ‘task-node’.

After generating the task-node tree, the system starts driving forward (upward in the tree), from the child-nodes to the parent, transfer the results from the child-nodes to their parents and updating the rules which correspond to the parents. this operation will continue until the top of the tree is reached (the task-node). The results of the intermediary nodes are broadcast to RB, and they are stored in the flags of the corresponding rules premises. The reasons behind using the rules to store the results of the nodes are:

1. To keep the degree of communication amongst the processors to a minimum.
2. An easy way to find out whether the goal is achieved or not.
3. Enable the system to explain its reasoning process, i.e. how conclusions are reached, what path is followed to achieve the goal, etc..
4. To keep each processor informed of the results produced by the other running processors. The processors exchange information about the search process through RB, and more specifically, by using the flags in the rules.
Figure 6.14: Driving Backward and Forward in RB, i.e. Downward and Upward in the search tree TON
5. The Get-Result (GR) module

The main job of GR is to conclude the result of the consultation. To achieve this task, the module cycles through the following phases (see Figure 6.17):

**Phase 1:**

IF the goal was achieved THEN

- Broadcast the result to the SRE module.
- Find the ID of the processor which proved the goal. This is stored in the field 'processing-id' of the solved goal node in ONS.
- Transfer TON [processing-id] to SRE. This is the search tree generated by the processor, while trying to achieve the goal. Thus, it tells us about how the goal was achieved, and the path that links the goal node to the terminal nodes.

**Phase 2:**

IF the goal was not achieved by PS THEN

Step 1 Collect all the solved nodes in ONS. Add these nodes to a set 'S'.

Step 2 IF S is empty THEN the goal is not achieved.

Step 3 ELSE {S is not empty}.

S1 Transfer the solvability of the nodes in S to the rules flags:

FOR each node (Snode) in S DO

{ Begin FOR }

- Scan RB to find all the rules that have Snode in their left-hand sides (premise part). All the candidate rules are maintained in a set called 'SetRule'.

285
• FOR every rule $r_i$ in ‘SetRule’ DO
  
  • Locate the premise which corresponds to Snode, i.e.
    $(\text{premise} = \text{Snod.e.node})$.

  • Set the premise’s flag to TRUE.

  {End FOR}

S2 Find the maximum level of the nodes in $S$ (Call this $\text{level-Max}$). This implies that we have to check the level field in every node in $S$ and conclude the maximum of the list of all the nodes levels.

S3 Generate the goal tree up to the $\text{level-Max}$ level, i.e. only generate the nodes which have their levels less than or equal to $\text{level-Max}$ (see Figure 6.15). The algorithm for this step is:

1. Start with the goal node ‘G’. ($\text{level-N} = 0$)

2. Generate all its choice-nodes:
   
   • Add the node to TON

   • Add its choice-nodes to choice-list.

3. Expand the node and its choice-nodes, and increment $\text{level-N}$ by one.
   
   • Add the child-nodes of the node to TON. Link the child-nodes to their parents.

   • Add the child-nodes of each choice-node to TON. Link the child-nodes to their parents.

   • Transfer the results from the flags of the rules that correspond to the parent and the choice-nodes, to the child-nodes.

4. Increment $\text{level-N}$ by one.
5. Expand the new added nodes (child-nodes) if possible:

- FOR each of the new added child-nodes (C-node) DO
  - Generate the choice-nodes of C-node, and add them to its choice-list.
  - IF (level-N ≤ level-Max) THEN
    - Generate the child-nodes of C-node.
    - Generate the child-nodes of the choice-nodes.
    - Link the new child-nodes to their parents.
    - Transfer the results from the flags of the appropriate rules to the new added child-nodes (rules that corresponds to the parents of the child-nodes).

6. IF we reach level-Max THEN stop, otherwise goto 4 and repeat the same process for the new generated child-nodes.

S4 Suppose that the number of nodes in TON is equal to Numb-Node.

S5 Drive upward in the tree, from the child-node to their parent-nodes, in order to transfer the results from the child-nodes to their parent-nodes. Update the flags of the corresponding rules accordingly. Thus, the results are broadcasted from the nodes in one level to the nodes in the previous level (see Figure 6.16). The algorithm for this step is:

1. Start with the last node in TON.
   position ← Numb-Node

2. GoTo the parent of the node. (Call the parent-node node-P).

3. Get the number rule of the rule which corresponds to node-P.
rule\_I \leftarrow \text{node-}P. \text{ rulenb}

4. GoTo the rule with index rule\_I in RB. \{\text{rule} \leftarrow \text{RB}[\text{rule}\_I]\}.

5. Update the flags in the premises of the rule according to the solvability of the child-nodes of node-\text{P}. Note that each child-node corresponds to a premise in the rule. The algorithm for this step is:

a. \text{FOR each child (C-node) of node-\text{P} DO} \\
\{\text{Begin FOR}\}

\begin{itemize}
\item Check if C-node or any of its choice-nodes is solved, and if so, mark the OR-parallel node as solved (C-node.result = 1).
\item Update the corresponding premise in rule (set the flag to TRUE).
\item IF neither C-node nor any of its choice-node is solved, then mark the OR-parallel node as not solved (C-node.result = -1). Update the corresponding premise in rule (set the flag to FALSE).
\end{itemize}
\{\text{End FOR}\}

b. IF all the OR-parallel nodes corresponding to the child nodes are solved (result = 1), then mark the parent-node solved, and mark the OR-parallel node corresponding to node-\text{P} as solved.

\{\text{node-\text{P}. result} \leftarrow 1; \text{node-\text{P}. solved} \leftarrow 1\}

c. IF node-\text{P} has some choice-nodes (choice-list is not empty) THEN do the same for the child-nodes of each of the choice-nodes. The algorithm for this step is:

\text{FOR every choice-node (choice-N) in node-\text{P} choice-list DO} \\
\begin{itemize}
\item Get the number rule\_I of the rule which corresponds to ‘choice-N’.
\end{itemize}
• FOR each child (C-node) of 'choice-N’ DO

{Begin FOR}

- Check if C-node or any of its choice-nodes is solved, and if so,
  mark the OR-parallel node corresponding to C-node as solved
  (C-node.result = 1). Update the corresponding premise in the
  rule RB[rule_I] (rule with number rule_I in RB).
- IF neither C-node nor any of its choice-node is solved, then
  mark the OR-parallel node as not solved (C-node.result = -1).
  Update the corresponding premise in the rule numbered rule_I
  in RB (set the flag to FALSE).

{End FOR}

• IF all the OR-parallel nodes corresponding to the child nodes are
  solved (result = 1 for each OR-parallel node), then mark the
  parent-node solved, and mark the OR-parallel node corresponding
  to the parent-node as solved.

\{node-P.result ← 1; node-P.solved ← 1\}.

6. IF neither node-P is solved nor any of its choice-nodes is solved, then
mark the corresponding OR-parallel node as not solved.
\{node-P.result = -1\}.

7. Move upward in TON to the node which comes just before the first
child of node-P, i.e. the node with address (node-P.first -1) in TON.
\{position ← node-P.first - 1 \}.

8. Repeat (2 to 7) until we reach the top node in TON (the goal node).

Thus:
IF position > 1 Go To 2

ELSE Go To the next step (S6).

S6 Conclude the result of the consultation. This is the status of the OR-parallel node corresponding to the goal node, which is the top node in TON (position = 1).

S7 Broadcast the result of the consultation to SRE module.

---

Figure 6.15: The tree corresponding to step (S3)

---

Figure 6.16: Transfer of results between the different levels of a tree (step S5)
6. The Show Result and Explain (SRE) module

The main job of this module is to give the result of the consultation, and provide the user with the necessary explanation, as about how the system has
reached certain results, etc., if this is needed.

The module cycles through the following phases:

**Phase 1:** Check the information broadcast by GR module. IF the goal was not achieved THEN give out a failure message.

**Phase 2:** IF the goal was achieved THEN two cases are possible:

*Case 1:* The goal was achieved during the PS module run-time, by one of the processors. The processor ID (the processor which proved the goal), and the corresponding search tree have been transferred to SRE by GR. Assume that the processor ID is equal to \( \ell (0 < \ell \leq q) \), thus the search tree is represented by TON[\( \ell \)]. Since the goal is achieved, this means that there is a 'valid path' which links the goal node to the terminal nodes in the search tree TON[\( \ell \)]. This 'valid path' is defined by the solved OR-parallel nodes, and the solved normal nodes in TON[\( \ell \)].

Driving downward in the search tree, from a solved node to its children, can be seen as driving backward in RB, from the conclusion of a rule to its premises. The node corresponds to one rule, and its child-nodes correspond to the premises of the rule. Driving upward in the search tree, from the child-nodes to the parent-node, can be seen as driving forward in RB, from the premises of the corresponding rule to its conclusion.

The algorithm for Case 1 is:

**Step 1** Give the result of the consultation.

**Step 2** IF explanation is needed THEN

1. Start with the top node in TON[\( \ell \)], the goal node. Add the corresponding rule's number to a list list-rule. This list will be used to maintain
the applicable rules numbers. An applicable rule corresponds to a solved node. In other words, if a rule $r_i$ corresponds to a node $node_j$, $r_i$ is said to be applicable, only if $node_j$ is solved.

2. Go To the next node in TON [$\ell$]. \( \{index \leftarrow 2\} \).

3. WHILE there are still non-visited nodes in TON [$\ell$], i.e. $index \leq$ Number of nodes in TON [$\ell$], DO

- Visit the current node, node with address in TON [$\ell$] equal to $index$.
- IF the node is solved \{node.result = 1\} THEN
  - IF the node is an OR-parallel node THEN check which one of the OR-nodes is solved and add the corresponding rule's number to the front of list-rule. To find the solved OR-node, we check the node in TON [$\ell$] and the choice-nodes in choice-list, we locate the solved node \{node.solved = 1\} and subsequently the rule number is 'node.rulenb'.
  - IF the node is a normal node, i.e. choice-list is empty then add the corresponding rule's number to the front of list-rule.
- IF the node is terminal or not solved THEN do nothing.
- Advance to the next node in TON [$\ell$], increment $index$.

At the end of Step 2 list-rule contains all the applicable rules, SRE will read the rules in list-rule one by one and give the necessary explanation.

Note that the explanation could have been given by first collecting the applicable rules in RB. (This is done by checking the flags of a rule and verify that they are all set to TRUE), and then using these rules to give the necessary explanation.
Case 2: The goal was achieved during the GR module run-time. The results of the search (the reasoning followed to prove the goal) are stored in the premises flags of the rules. For each solved node in the search tree corresponds an applicable rule in RB. To give the explanation about how the goal was proved, SER can follow one of two strategies:

Strategy 1: Collect all the applicable rules in RB. This is done by simply checking the flags of the rules. If, for a rule, all the flags are set to TRUE then the rule is applicable. The SER module then uses the applicable rules to produce the appropriate explanation.

Strategy 2: In strategy 2, SER gives the explanation by generating the 'valid path' which proves the goal. This is the path in the goal search tree, which links successfully the goal node to the terminal nodes. To generate the 'valid path' SER starts with the goal node, expands it by reading the information stored in the flags of the corresponding rules in RB. This process is then repeated for the child-nodes, and then for their children, and so on, until the whole path is generated. The general algorithm for strategy 2 is:

1. Start with the goal node. Add it to TON, and initialize position to 1.
2. Scan RB to find the corresponding applicable rule. This is the rule with the node in its conclusion part, and with all its premises flags set to TRUE.
3. Add the rule’s number rule_I to the front of a list list-rule. This list maintains the numbers of the applicable rules corresponding to the solved nodes in the 'valid path'. Set the 'rulenb' field in the node with rule_I.
4. Expand the node generating its child-nodes. Add them to TON.

5. Advance to the next node in TON. Increment position by one.

6. Repeat 2 to 5 until no more expansion of nodes is possible, i.e. all the nodes in TON have been explored and no new nodes are added to TON.

At this point the 'valid path' is fully generated and the list list-rule contains all the applicable rules corresponding to the solved nodes in the path. The SER module reads list-rule and produces the appropriate explanation.

6.4.2 PERFORMANCE ANALYSIS OF PBC-D1

The models have been implemented on the Sequent Balance 8000, shared memory multiprocessor system and were tested for a Rule-based Expert System, with different numbers of rules, so as to study the effects of the problem sizes and complexity on speed-up and efficiency. The experiments are carried out with different numbers of processors, so as to find out the speed up obtained from each of the parallel implementations.

Let \( T(p) \) be the time spent by \( p \) processors to execute a given algorithm and \( T(1) \) be the time-complexity of the same algorithm running on one processor. Then, the speed-up \( (Sp) \) is defined as:

\[
Sp = \text{SPEED UP} = \frac{T(1)}{T(p)} \leq p
\]

and the efficiency \( (E) \) is defined as:

\[
E = \text{EFFICIENCY} = \frac{Sp}{p} = \frac{T(1)}{p \cdot T(p)} \leq 1.
\]
For performance measurement reasons, we do not time the initial loading and set up of the knowledge base (rules and facts) in any of the parallel models. Only the process of inferencing is timed.

The results from our tests of the PBC-D1 model are summarized in Tables 6.1, 6.2 and 6.3.

The graphical interpretations of these tables are shown in Figures 6.18, 6.19 and 6.20 respectively.

As the tables show, the experiments were carried out using three sets of rules with different sizes, i.e. different numbers of rules. This is to study the effect of the problem size and complexity on the performance of the parallel model in the execution of expert systems.

It can be observed from the figures that the speed curve follows more closely a log$_p$ type curve (where $p$ is the number of processors used). When $p$ satisfies a small number of processors (1 to 9). This is due to the fact that the bulk of the processing operations are involved with searching and matching. In addition, extensive rulebase and database access by the processors involving communication overheads on the shared memory bus are involved. It is well known that the non-numerical operations of searching and matching are more difficult to parallelize than the standard scientific problems. This effect is even more pronounced with a smaller size of the problem as shown in experiment 1. Finally, it can be observed from the experiments that the efficiency is better when the size of the rulebase is larger. On the other hand, the efficiency index for smaller number of processors (1 to 5) indicates an acceptable performance level, however for larger number of processors the converse is true.
<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Computing Time (msec)</th>
<th>SPEEDUP</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>120500</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>62400</td>
<td>1.931</td>
<td>0.966</td>
</tr>
<tr>
<td>3</td>
<td>43600</td>
<td>2.164</td>
<td>0.921</td>
</tr>
<tr>
<td>4</td>
<td>36090</td>
<td>3.339</td>
<td>0.835</td>
</tr>
<tr>
<td>5</td>
<td>29970</td>
<td>4.021</td>
<td>0.804</td>
</tr>
<tr>
<td>6</td>
<td>27460</td>
<td>4.388</td>
<td>0.731</td>
</tr>
<tr>
<td>7</td>
<td>26400</td>
<td>4.564</td>
<td>0.652</td>
</tr>
<tr>
<td>8</td>
<td>24900</td>
<td>4.839</td>
<td>0.605</td>
</tr>
<tr>
<td>9</td>
<td>23870</td>
<td>5.048</td>
<td>0.561</td>
</tr>
</tbody>
</table>

Table 6.1: Experimental Results for the Parallel Model PBC-D1. [Rulebase = 250 rules].

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Computing Time (msec)</th>
<th>SPEEDUP</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>203040</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>107150</td>
<td>1.895</td>
<td>0.947</td>
</tr>
<tr>
<td>3</td>
<td>74630</td>
<td>2.721</td>
<td>0.907</td>
</tr>
<tr>
<td>4</td>
<td>60190</td>
<td>3.373</td>
<td>0.843</td>
</tr>
<tr>
<td>5</td>
<td>52400</td>
<td>3.875</td>
<td>0.775</td>
</tr>
<tr>
<td>6</td>
<td>46960</td>
<td>4.324</td>
<td>0.721</td>
</tr>
<tr>
<td>7</td>
<td>44510</td>
<td>4.562</td>
<td>0.652</td>
</tr>
<tr>
<td>8</td>
<td>42130</td>
<td>4.819</td>
<td>0.602</td>
</tr>
<tr>
<td>9</td>
<td>40170</td>
<td>5.055</td>
<td>0.565</td>
</tr>
</tbody>
</table>

Table 6.2: Experimental Results for the Parallel Model PBC-D1. [Rulebase = 320 rules].
<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Computing Time (msec)</th>
<th>SPEEDUP</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>248320</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>127850</td>
<td>1.942</td>
<td>0.971</td>
</tr>
<tr>
<td>3</td>
<td>87710</td>
<td>2.849</td>
<td>0.950</td>
</tr>
<tr>
<td>4</td>
<td>67960</td>
<td>3.654</td>
<td>0.913</td>
</tr>
<tr>
<td>5</td>
<td>62880</td>
<td>4.349</td>
<td>0.790</td>
</tr>
<tr>
<td>6</td>
<td>53030</td>
<td>4.683</td>
<td>0.780</td>
</tr>
<tr>
<td>7</td>
<td>48800</td>
<td>5.069</td>
<td>0.727</td>
</tr>
<tr>
<td>8</td>
<td>47460</td>
<td>5.232</td>
<td>0.654</td>
</tr>
<tr>
<td>9</td>
<td>45980</td>
<td>5.401</td>
<td>0.600</td>
</tr>
</tbody>
</table>

Table 6.3: Experimental Results for the Parallel Model PBC-D1. 
[Rulebase = 450 rules].

Figure 6.18: Speed-up graph for the model PBC-D1. 
[Rulebase = 250 rules].
Figure 6.19: Speed-up graph for the model PBC-D1.
[Rulebase = 320 rules]

Figure 6.20: Speed-up graph for the model PBC-D1.
[Rulebase = 450 rules]
6.4.3 THE PBC-D2 MODEL

The PBC-D2 model uses the same data structures used for PBC-D1. These are:

1. The Rulebase (RB).
2. The Database (DB).
3. The OR-Node-Set (ONS).
4. The Tree-Of-Nodes (TON).
5. The Processors-Status-Array (PSA).

In addition to these structures PBC-D2 uses a new data structure called \textit{wait-request} array. This has the same structure used for PSA. Thus \textit{wait-request} is an array of Boolean with \( p \) elements, where \( p \) is the number of processors used by the system.

This array structure is used in the following way:

For a processor \( P_i \) we have the following:

1. \( \text{wait-request} [P_i] = \text{FALSE} \): The processor \( P_i \) is in a wait status, but has not requested a job yet. In this case \( P_i \) is allowed to initiate a job-request.
2. \( \text{wait-request} [P_i] = \text{TRUE} \): The processor is in a wait status, and has already requested a job. In this case the processor is not allowed to make any more request. Thus \( P_i \) has to wait for the requested job to become available.

A processor requests a job by calling the parallel programming library routine 'm-next'. This call returns the value of a global counter (an integer value), and then increments this counter by one. Thus, if the first call to 'm-next' returns the integer value \( \lambda \), then the next call will return the value \( (\lambda + 1) \), and so on.

Assume that a processor's call to 'm-next' returns an integer value \( N \).
The requested job \((T_N)\) is defined as:

\[
T_N = \text{Process the OR-Node with the address in ONS is equal to } N.
\]

In other words,

\[
T_N = \text{Process } (\text{ONS}[N]).
\]

The model consists of 6 main modules:

1. The User-Interface (UI) module.
2. The Initialize-Flags (IF) module.
3. The Backward-Chaining (BC) module.
4. The Parallel-Search (PS) module.
5. The Get-Results (GR) module.
6. The Show-Results-Explain (SRE) module.

The concept of PBC-D2 is depicted in Figure 6.21.

Figure 6.21: The PBC-D2 model concept
We assume that:

a) The rules are numbered from 1 to n.

b) The processors are numbered from 1 to m.

c) The facts needed by the system throughout its reasoning process are collected by the User-Interface module at the beginning of the consultation. This is done for performance measurement reasons.

The three modules: UI, GR and SRE are kept the same as before in PBC-D1.

The other modules are modified to accommodate the new parallel strategy used in PBC-D2. The modifications and the parallel strategy are explained in the following sections.

As in PBC-D1, ONS is used to implement the concept of the 'task-queue' in PBC-D2. A task consists of investigating a number \( \ell \) of nodes from ONS \( (\ell \geq 1) \).

**Definition 9:**

Let \( T \) be a task which consists of investigating \( \ell \) nodes \( (\ell \geq 1) \). node_1, node_2, \ldots, node_{\ell-1}, node_{\ell} \) from ONS:

1. The 'Base' of the task \( T \) \{ Base(T) \} is defined as the index of node_1 in ONS.
2. The 'Top' of the task \( T \) \{ Top(T) \} is defined as the index of node_{\ell} in ONS.
3. Processing the task \( T \) is defined as investigating the nodes between Base(T) and Top(T), in ONS.
4. The 'Size' of \( T \) is defined as equal to the number of nodes that the task \( T \) has to investigate. Thus \( Size(T) = \ell \).
Definition 10:

Let $P_i$ be a processor. A 'Bite' of work for $P_i$ is defined as the number of tasks assigned to $P_i$ at any one time. Thus Bite is equal to the number of tasks to be assigned to $P_i$ at one time, i.e. in one go.

Rq 1: The variable 'Bite' is considered to be equal to one task in our discussion here. This means that the job assigned to each processor at each cycle, consists of only one task to process. The terms 'Bite' and 'Task' are sometimes used interchangeably to mean the same thing.

Rq 2: In our discussion here, we assume that the number of nodes which forms a task $T$ is equal to one. In other words, the 'Size' of a task is considered to be equal to one. Thus,

$$\forall T \in \text{Task-queue}; \text{Size}(T) = 1.$$

and,

$$\text{Processing}(T) \equiv \text{Processing the node defined by } T.$$

1. The Initialize-Flags (IF) module

The main job for this module is to initialize the flags of the rules premises in accordance with the initial facts in DB. It matches the rules premises against the initial facts in DB, and then set the premises to either TRUE or FALSE according to the result of the match test. The algorithm for this module is:

Step 1 Define the number of processors to be used to execute the job in parallel. (Call this number $q/1 \leq q \leq m$).

Step 2 Set up the $q$ processors for work.

Step 3 Create $q$ processes to execute the job.
Step 4 Execute the job (Initialize the flags) in parallel. This is done using data-partitioning with a dynamic scheduling technique. The algorithm for this step is as follows (See Figure 6.22):

1. FOR each process DO:

   a. Call 'm-get-numprocs' to determine the number of processes created for the job. (This number is q).

   b. Call 'm-next' to find out which task to process. This call returns an integer value \( \lambda (\lambda \geq 1) \). The first call to 'm-next' returns the value 1, the second returns 2, and so on. This function is used by the processes to determine their tasks in a dynamic fashion. RB is considered by the processes as a task-queue. Each task consists of initializing the flags of a number \( \ell \) of rules. In our discussion here \( \ell \) is considered to be equal to one. Thus a process’s task is to initialize the flags of only one rule. The rule number is determined using ‘m-next’. (Bite = 1, Size = 1).

   c. Determine the task \( T_\lambda \) to carry out:

   \[
   \text{Base} (T_\lambda) = \text{Bite} \ast \text{m-next} = \lambda \\
   \text{Top} (T_\lambda) = \text{Base} + \text{Bite} - 1 = \lambda + 1 - 1 = \lambda.
   \]

   d. Process the task \( T_\lambda \), i.e. Initialize the flags in the premises of the rule \( \text{rule}_\lambda (\text{rule}_\lambda = \text{RB}[\lambda]) \). The algorithm for this step is the following:

   - IF \( \lambda > n \) THEN no work is done and the process becomes idle.
   - ELSE

     o Process the rule \( \text{rule}_\lambda \):

     304
{FOR each condition $C_\ell$ in the premise part of 'ruleA' DO

- Match $C_\ell$ against the facts in DB.
- IF $C_\ell$ is satisfied THEN set its flag to TRUE otherwise, set the flag to FALSE }.

e. IF there is more work to do, i.e. more tasks in the 'task-queue' THEN

GoTo Step 4(1.a).

Step 5 Terminate all the processes.

2. The Backward Chaining (BC) module

This organizes and controls the overall reasoning process. It supervises the PS, GR, and SRE models and coordinates their activities. The BC module cycles through the following phases (see Figure 6.23):
Phase 1:

Scan the RB to find all the rules that have the goal in their right-hand side. The candidate rules are then maintained in an array called \textit{rule-set}. If \textit{rule-set} is empty, then no appropriate rule is found and consequently the goal is not valid and a failure message is send. Otherwise, start Phase 2.

Phase 2:

The algorithm for this phase is:

**Step 1** Check if the goal exist in DB. If so, the goal is achieved, otherwise we start Step 2.

**Step 2**

1. Define the number of processors to be used to execute the job in parallel. Assume that this number is \( q \) \( (1 \leq q \leq m) \)

2. Define the variable ‘Bite’, i.e. the number of tasks to be assigned to one processor at each cycle of the parallel search. As explained before, this is the number of OR-nodes in ONS to be assigned to one processor at any one time. Note that in our discussion here this number is considered as equal to one. Thus, each processor is assigned only one task at each job-request (see Figure 6.24) \{Bite = One task\}.

3. Set up the \( q \) processors for work.

4. Initialize all the elements of PSA to TRUE. This indicates that, at start, all the processors are waiting. Initialize all the elements of \textit{wait-request} array to FALSE, to indicate that the processors are not in a wait-request status at the start of the consultation.
Scan the rulebase to find the rules that have the goal in their right-hand sides. Store these rules in 'rule-set'.

- is rule-set empty ??
  Yes: Stop and send a failure message
  No:
    - goal \( \in \) DB ??
      Yes: goal is achieved (Trivial Case)
      No: Define the number of processors to use for the job

Define the variable 'Bite', which is the number of tasks to be assigned to one processor at one time

Set up the processors for work

Initialize PSA elements to TRUE, i.e. Initialize the status of the processors to wait. Initialize the elements of 'wait-request' array to FALSE; processors are not in a wait-request status at start

Add the nodes corresponding to the rules in 'rule-set' to ONS

Initialize the variables 'Halt' and 'Solfound' to FALSE

Activate the PS module (figure 6.25)

Terminate the processes

Activate the GR module and get the result of the consultation

Activate the SRE module

END

Figure 6.23: The Backward chaining module in PBC-D2
5. Add the nodes corresponding to the rules in rule-set to ONS, i.e. add to ONS the OR-parallel nodes corresponding to the goal.

6. Initialize the variables ‘Solfound’ and ‘Halt’ to FALSE.

7. Start the next module PS, which assigns the backward chaining process to the q processors to execute it concurrently.

8. Broadcast the results of processing the tasks (OR-nodes) in ONS to the GR module, which then concludes the result of the consultation.

9. Start the module SRE, which shows the result and give necessary explanations, if needed.

3. The Parallel Search (PS) module

The only difference between this module and the PS module used for PBC-D1 is in the way the processors schedule their tasks. The tasks are partitioned among the running processors in a dynamic fashion implemented by using the ‘m-next’ parallel library routine. The new scheduling strategy is depicted in Figure 6.25. The scheduling algorithm is as follows:

Each processor cycles through the following steps:

- (a) $P_i$ requests a task. ($\text{status} = \text{WAIT}$)
- (b) $P_i$ is assigned only one task. ($\text{status} = \text{ACTIVE}$)

![Diagram of task assignment](image)

**Figure 6.24:** The assignment of tasks to a processor $P_i$
(1) Get the processor ID. (Call this proc-id).

(2) WHILE (Not Halt) and (Not SolFound) DO

{Begin WHILE}

a) Call ‘m-get-numprocs’ to find out the number of processes created for the job.

b) IF the processor is not in a wait-request status THEN

- Call ‘m-next’ to determine the task to perform. This call returns an integer value λ (λ ≥ 1).

c) Start a critical section (lock a LOCK).

d) IF λ is greater than the number of nodes in ONS (Numb-OR) THEN

the task is not available in the task-queue ONS, consequently the process behaves as follows:

- Wait, i.e. change the process’s status to WAIT.

- Check the status of the other processes.

- IF they are all waiting (in a wait status) THEN

  - Halt the execution, and the job is finished. (Halt ← TRUE).

- ELSE (not all the processors are waiting)

  - The processor makes a job-request:

    \[ wait-request[proc-id] ← TRUE. \]

  - Exit the critical section. Unlock the LOCK.

e) ELSE {task is available}

- Define the task to perform, i.e. define the ‘Base’ and ‘Top’ of the task. This is done as follows:
$T_\lambda$ is such as:

$$\text{Base } (T_\lambda) = \text{Bite } \ast m-\text{next } = 1 \ast \lambda = \lambda$$

$$\text{Top } (T_\lambda) = \text{Base } + \text{Bite } - 1 = \lambda + 1 - 1 = \lambda.$$  

Thus $T_\lambda$ is defined as: “process the node ONS [\lambda].”

- Collect the task $T_\lambda$. (This is ONS[\lambda]).
- Change the process’s status to ACTIVE.
- The processor is no longer in a wait-request status;

  $$\text{wait-request } [\text{proc-id}] \leftarrow \text{FALSE}.$$  

- Process the task. This action might generate new tasks (OR-nodes).
  
  If so:
  - Start a critical section.
  - Add the new tasks to the end of the task-queue, i.e. add the new generated OR-nodes to the end of ONS, and update the number of OR-nodes $\text{Numb-OR}$ accordingly.
  - Exit the critical section.

{End WHILE}

6.4.4 PERFORMANCE ANALYSIS OF PBC-D2

Tables 6.4, 6.5 and 6.6 show the experimental results obtained for the parallel model PBC-D2. The graphical interpretations of these tables are shown in Figures 6.26, 6.27 and 6.28 respectively.

Again, the experiments were carried out using three sets of rules with different sizes, i.e. different number of rules.

It can be observed from these results that there is a slightly improvement in the efficiency for the PBC-D2. Therefore similar conclusion can be drawn as indicated previously.
Get the processor ID (call this 'proc-id')

While (not SolFound) and (not Halt) DO

Begin WHILE

Is the processor in a wait-request status, i.e. wait-request [proc-id] = TRUE

YES

Make a request for a job while waiting. Call 'm-next' to determine the task to perform. This call returns an integer value \( \lambda \geq 1 \)

Start a critical section. Lock a LOCK

Is the task available in the task queue? Is \( \lambda \leq \) number of nodes in ONS ??

YES

The processor is no longer in a wait-request status. wait-request [proc-id] <- FALSE

Process the task (Fig. 6.11)

NO

The job is completed. halt the execution. Halt <- TRUE

Are all the other processors waiting ??

YES

End WHILE

Exit the critical section unlock the LOCK

NO

Exit the critical section unlock the LOCK

Figure 6.25: The scheduling technique used in PBC-D2
<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Computing Time (msec)</th>
<th>SPEEDUP</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>120820</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>62620</td>
<td>1.929</td>
<td>0.965</td>
</tr>
<tr>
<td>3</td>
<td>43770</td>
<td>2.760</td>
<td>0.920</td>
</tr>
<tr>
<td>4</td>
<td>36440</td>
<td>3.316</td>
<td>0.829</td>
</tr>
<tr>
<td>5</td>
<td>29890</td>
<td>4.042</td>
<td>0.808</td>
</tr>
<tr>
<td>6</td>
<td>27330</td>
<td>4.421</td>
<td>0.737</td>
</tr>
<tr>
<td>7</td>
<td>26020</td>
<td>4.643</td>
<td>0.663</td>
</tr>
<tr>
<td>8</td>
<td>25030</td>
<td>4.827</td>
<td>0.603</td>
</tr>
<tr>
<td>9</td>
<td>24000</td>
<td>5.034</td>
<td>0.559</td>
</tr>
</tbody>
</table>

Table 6.4: Experimental Results for the Parallel Model PBC-D2. [Rulebase = 250 rules].

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Computing Time (msec)</th>
<th>SPEEDUP</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>203110</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>107170</td>
<td>1.895</td>
<td>0.948</td>
</tr>
<tr>
<td>3</td>
<td>74870</td>
<td>2.713</td>
<td>0.904</td>
</tr>
<tr>
<td>4</td>
<td>60800</td>
<td>3.341</td>
<td>0.835</td>
</tr>
<tr>
<td>5</td>
<td>52440</td>
<td>3.873</td>
<td>0.775</td>
</tr>
<tr>
<td>6</td>
<td>47080</td>
<td>4.314</td>
<td>0.719</td>
</tr>
<tr>
<td>7</td>
<td>44210</td>
<td>4.594</td>
<td>0.656</td>
</tr>
<tr>
<td>8</td>
<td>41940</td>
<td>4.843</td>
<td>0.605</td>
</tr>
<tr>
<td>9</td>
<td>39940</td>
<td>5.085</td>
<td>0.565</td>
</tr>
</tbody>
</table>

Table 6.5: Experimental Results for the Parallel Model PBC-D2. [Rulebase = 320 rules].
Table 6.6: Experimental Results for the Parallel Model PBC-D2.

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>Computing Time (msec)</th>
<th>SPEEDUP</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>247980</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>128010</td>
<td>1.937</td>
<td>0.969</td>
</tr>
<tr>
<td>3</td>
<td>89190</td>
<td>2.780</td>
<td>0.927</td>
</tr>
<tr>
<td>4</td>
<td>68400</td>
<td>3.625</td>
<td>0.906</td>
</tr>
<tr>
<td>5</td>
<td>58410</td>
<td>4.246</td>
<td>0.849</td>
</tr>
<tr>
<td>6</td>
<td>52870</td>
<td>4.690</td>
<td>0.782</td>
</tr>
<tr>
<td>7</td>
<td>48000</td>
<td>5.166</td>
<td>0.738</td>
</tr>
<tr>
<td>8</td>
<td>46800</td>
<td>5.299</td>
<td>0.662</td>
</tr>
<tr>
<td>9</td>
<td>46230</td>
<td>5.364</td>
<td>0.596</td>
</tr>
</tbody>
</table>

Figure 6.26: Speed-up graph for the model PBC-D2.

[Rulebase = 450 rules]

Figure 6.26: Speed-up graph for the model PBC-D2.

[Rulebase = 250 rules]
Figure 6.27: Speed-up graph for the model PBC-D2.  
[Rulebase = 320 rules]

Figure 6.28: Speed-up graph for the model PBC-D2.  
[Rulebase = 450 rules]
6.5 Further Work and Conclusion

In this chapter we have presented two parallel Backward Chaining models, together with their implementation and the analysis of their performance. In this section we will present some further work and address some difficulties encountered during the implementation stage.

(I) Ideas for further work

In the parallel Backward Chaining models presented in this chapter, the condition for a processor to be able to split up its search tree is that there is at least one processor in a waiting status. This is done, to keep all the processors as busy as possible and therefore reduce their idle time.

Moreover, the processor splits up its search tree, only if there is another processor ready to search the resulting subtree. Thus, a new task is generated only if requested by a waiting processor. This strategy provides a good control on the number of the tasks in the task-queue and guarantees a coarser granularity for the tasks. The two models are suited to coarse-grained parallelism: The subtree assigned to each processor (the 'task' of the processor) needs to be reasonably large in order to allow the benefits of parallelism to outweigh the overheads resulting from distributing and allocating the tasks. It is thought that allowing every processor to split up its search tree at every choice point (OR-parallel node) will make the parallelism too fine-grained. To study this idea further and develop a good understanding about the effect of the size (granularity) and the number of the tasks on the performance of the backward chaining process we suggest the following ideas:
Idea 1: *(Control based on the level of the OR-parallel nodes in the goal tree).*

In this method, the condition for a processor to split up its search tree and generate new tasks (subtree to be processed by other processors) is related to the level of the 'task-nodes' in the goal search tree. The OR-nodes are divided into two classes: 'normal' OR-nodes, and 'OR-parallel' nodes. A 'normal' OR-node is processed by one processor only. Thus the node and its choice nodes are added to TON. An 'OR-parallel' node is processed by more than one processor. The choice-nodes are considered as tasks and are added to the task-queue 'ONS' to be assigned to different processors. Each node is represented by a record which has a field called 'level', that tells about the level in the tree at which the node is found. The 'level' of a node is the level in the whole 'goal search tree' where the node is found (see Figure 6.29).

![Figure 6.29: The concept of levels in the goal tree](image-url)
In this Figure the node $X$ is found at level 3.

The condition for an OR-node to be considered as 'OR-parallel' node is that the level of this node is 'less than or equal to the value of a global variable 'level-G', which is defined by the user at the beginning of the consultation. Thus, a processor is allowed to split up its search tree, at a choice point (OR-node), only if the level of the OR-node is less than or equal to 'level-G'.

To implement this idea, the following changes have to be made to PBC-D1 and PBC-D2:

1. The backward chaining (BC) module reads in the value of 'level-G' which is defined by the user.

2. In the PS module the decision on whether to add the choice-nodes of an OR-node to TON (the node is then a normal OR-node) or to ONS (the node is allowed to be an OR-parallel node) is now based on how the level of the node compares to 'level-G'. Steps 11 and 12 in PS have to be replaced by the following steps 11a and 12a:

**Step 11a IF** (nodeC.level > level-G) **THEN**

- The processor is not allowed to split up its task 'processing nodeC', and the choice-nodes are added to the choice-list of TON [processor] to be processed by the same processor.

**Step 12a IF** (nodeC.level ≤ level-G) **THEN**

- Split up the node tree and add the choice-nodes to the task-queue ONS.

This is done in the same way explained in Step 12.

This method provides the user with a better control over the partitioning of the goal search tree among the processors and the definition of the granularity of
the tasks in the task-queue. Coarse grained tasks are those corresponding to the OR-parallel nodes found at the first levels of the goal tree, (closest to the root of the tree). It is thought that, by keeping the value of ‘level-G’ small, the granularity of the tasks will be coarser, and consequently the parallelism will be better suited for shared-memory multiprocessor architecture.

This idea can be used as an experimental tool to study the following points:

1. The effect of the nodes level on the performance of the backward chaining process.

2. How the structure of the goal search tree can affect the performance of the parallel backward chaining process.

3. The relation between the granularity of the tasks and the performance of the parallel backward chaining model.

Idea 2: (Control based on the number of tasks in the task-queue ONS).

In this method, the condition for a processor to be able to split up its search tree and generate new tasks is related to the number of tasks in the task-queue. The idea is to introduce a variable called ‘Max-tasks’ and to keep the number of tasks in the task-queue, always less than ‘Max-tasks’. This variable can be defined in various ways. One way is to define ‘Max-tasks’ as a multiple of the number of processors allocated for the job. Thus, assuming that the number of processors allocated is \( q \) \((1 \leq q \leq \text{total number of processors on-line})\), ‘Max-tasks’ can be defined as equal to \( \lambda q \) where \( \lambda \) is an integer \((\lambda > 0)\). \( \lambda \) and \( q \) are defined by the user at the start of the consultation. We believe that, in order to achieve a better performance of the parallel model, the variable \( \lambda \) should be kept small.
Thus, a processor is allowed to split up its search tree, at a choice point (OR-node) and generate new tasks, only if the number of the tasks in the task-queue is less than 'Max-tasks'. The algorithm for this idea is the following:

1. The user defines the number of processors to be allocated for the job. (Call this number \( q \)).

2. The user defines the variable \( \lambda \).

3. 'Max-tasks' is equal to \( \lambda.q \).

4. FOR each processor processing a task \( T \) i.e. investigating the search tree (tree-\( T \)) of an OR-parallel node DO
   
   - At each choice point in ‘tree-\( T \)’, the processor compares the current number of tasks in the task-queue to ‘Max-tasks’.
   - IF (number of tasks is less than ‘Max-tasks’) THEN
     - Split up ‘tree \( T \)’ at the current choice point and generate new tasks.
   - IF (number of tasks is greater than or equal to ‘Max-tasks’) THEN
     - ‘tree-\( T \)’ is processed by the same processor.

This method provides the user with a good control on the number of OR-nodes in the task queue and an experimental tool to study the effect of the number of tasks on the overall performance of the backward chaining process.

(II) Implementation difficulties

The major difficulty encountered during the implementation stage of the parallel backward chaining models was the unavailability of the recursion for the child processes. A child process is not able to perform any recursive call. Thus a forked procedure should not include any recursive call in its body. This problem has led
to the use of more complicated data structure and additional use of queue structure
to replace the ‘recursion’ mechanism.
CHAPTER VII

Conclusion
The parallel models designed for the forward chaining and the backward chaining techniques respectively using static and dynamic scheduling for running concurrent processes, and with the database alternated between the shared main memory and the processors local memories were presented in Chapters 5 and 6.

By analysing the results obtained, one can conclude that the speed up curves follow more closely a $\log_p$ type curve (where $p$ is the number of processors used) when $p$ satisfies a small number of processors (1 to 9 processors). This is due to the fact that the bulk of the processing operations are involved with searching and matching. In addition, extensive Rulebase and Database access by the processors involving communications overheads on the shared memory bus are involved. However, it is well known that non-numerical operations of searching and matching are more difficult to parallelize than the standard scientific problems. This effect is even more pronounced with a 'smaller' size of the problem. It can be observed from the results that the efficiency is better when the size of the Rulebase is larger. On the other hand, the efficiency index for small numbers of processors (1 to 5 processors) indicates an acceptable performance level, however for larger number of processors the converse is true.

Another conclusion that can be drawn is that the results obtained (efficiency and speed up) when using the dynamic scheduling technique to allocate the tasks to the processors, are generally better than those obtained using the static scheduling technique. This is because in dynamic scheduling the computing load is more evenly distributed and no process is idle while there is still work to be done.

The use of local memories in the parallel forward chaining models clearly shows a large increase in performance. The speed up curve becomes almost linear, and
the efficiency of the models has greatly improved. This is resulting from the fact that the communication overheads are reduced when the matching is done in local memories. The processes' access to the shared memory is kept minimum and hence the results are better. This indicates that the use of local memory for parallel forward chaining system is a deciding factor in the design of parallel expert systems. A final conclusion is that the forward chaining technique 1 is more suitable for parallelisation than the forward chaining technique 2. Moreover both forward chaining techniques are easier to parallelize than the backward chaining technique.

However, because of the operation involved in expert system evaluation (in both the forward and backward chaining modes), these results represent an acceptable level of efficiency, It appears from the results that for the shared memory multiprocessor model the parallel strategies used in expert systems design are more favorable for small numbers rather than larger numbers of processors. In order to design an expert system to use a large number of processors efficiently, alternative strategies need to be further investigated. In addition, improvement is 'cache' design and larger local memories in each processor of the multiprocessor parallel computer will improve these results substantially.
References


325


[113] D. E. Shaw, *NON-VON's applicability to three AI tasks areas*, in Proc. 9th Int. Joint Conference on Artificial Intelligence, August 1985, pp. 61–70.


331


APPENDICES
(see accompanying disc)