A Parallel Process Model and Architecture for a Pure Logic Language

by

Innes E. Jelly MA, MSc

A thesis submitted to the Council for National Academic Awards in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

Sponsoring Establishment	:	School of Computing and Management Sciences, Sheffield City Polytechnic
Collaborating Establishment	:	ICL plc

October 1990

Contents

Chapter Two

Chapter Three

Chapter Four

Chapter Five

Chapter Seven

Chapter Eight

Chapter Nine

Appendices

Appendix A

Appendix B

Appendix C

Appendix D

Appendix E

Appendix F

Appendix G

Appendix H

Appendix I

Appendix J

List of Figures

Acknowledgments

This reseach project was funded under Project No. 086/178 by the Alvey Commision with ICL plc designated as industrial "uncle".

I should like to thank my Director of Studies, John Brown, for his interest, advice and many contributions to the project; my supervisor, Ian Morrey, for his support and encouragement; Ed Babb and his team at ICL for their interest and generous help; the research staff at Sheffield City Polytechnic for their companionship and advice, especially Jonathan Gray for his help with Transputer technology and Roger Spall for imparting his desktop publishing skills. Finally thanks are due to my husband and children for their patient support during the preparation of this thesis.

Abstract

The research presented in this thesis has been concerned with the use of parallel logic systems for the implementation of large knowledge bases. The thesis describes proposals for a parallel logic system based on a new logic programming language, the Pure Logic Language. The work has involved the definition and implementation of a new logic interpreter which incorporates the parallel execution of independent OR processes, and the specification and design of an appropriate non shared memory multiprocessor architecture.

The Pure Logic Language which is under development at ICL, Bracknell, differs from Prolog in its expressive powers and implementation. The resolution based Prolog approach is replaced by a rewrite rule technique which successively transforms expressions according to logical axioms and user defined rules until no further rewrites are possible.

A review of related work in the field of parallel logic language systems is presented. The thesis describes the different forms of parallelism within logic languages and discusses the decision to concentrate on the efficient implementation of OR parallelism. The parallel process model for the Pure Logic Language uses the same execution technique of rule rewriting but has been adapted to implement the creation of independent OR processes and the required message passing operations. The parallelism in the system is implemented automatically and, unlike many other parallel logic systems there are no explicit program annotations for the control of parallel execution. The spawning of processes involves computational overheads within the interpreter: these have been measured and results are presented.

The functional requirements of a multiprocessor architecture are discussed: shared memory machines are not scalable for large numbers of processing elements, but, with no shared memory, data needed by offspring processors must be copied from the parent or else recomputed. The thesis describes an optimised format for the copying of data between processors. Because a one-to-many communication pattern exits between parent and offspring processors a broadcast architecture is indicated. The development of a system based on the broadcasting of data packets represents a new approach to the parallel execution of logic languages and has led to the design of a novel bus based multiprocessor architecture. A simulation of this multiprocessor architecture has been produced and the parallel logic interpreter mapped onto it: this provides data on the predicted performance of the system. A detailed analysis of these results is presented and the implications for future developments to the proposed system are discussed.

Chapter One

Introduction

1.1. Introduction to the Project

The use of logic as a programming language developed out of work on automated theorem proving in the 1960s and 1970s: in recent years logic languages in particular Prolog have moved from being research tools to providing the facilities and performance expected from modern programming environments. However although the performance of many current Prolog systems has been improved with the introduction of sophisticated compiler techniques, the type of application in which Prolog is used often makes heavy computational demands on the system. This situation is typical of many programs employed in the field of artificial intelligence where extensive pattern matching operations are involved in the processing [Charniak 85]. Applications of this type include expert systems, natural language processing, deductive databases and other knowledge based systems [Frost 86].

The involvement of high computational demands in many logic language applications has led to research into the parallel execution of these programs. The underlying premise has been that by dividing the programming task into separate computational units which can be executed simultaneously, the overall performance can be improved. There has been a considerable amount of research into the definition of parallel logic languages and the design of suitable multiprocessor architectures, and this project contributes to the work in both of these areas.

The thesis describes the development of new proposals for a parallel logic system. The work has involved the definition and implementation of a new logic interpreter based on the parallel execution of independent processes, and the specification and design of a non shared memory multiprocessor computer for use with the parallel logic system. The manner in which the interpreter handles the creation and execution of independent processes differs from other parallel logic language implementations. The multiprocessor machine design represents a novel approach to the communication of data between different processing elements. A simulation of the combined system, ie the interpreter mapped onto the parallel architecture, has been produced and measurements on the predicted performance of the system obtained.

The project evolved out of work done at Sheffield City Polytechnic and the Systems Strategy Centre of ICL on parallel architectures and logic language execution and this is considered in the next section which presents the background to the project. This is followed by a discussion on the aims and development of the project. The final section of the chapter describes the organisation of the thesis.

1.2. The Project

1.2.1. Background to the Project

The starting point for this project has been research done at Sheffield City Polytechnic on the design of multiprocessor architectures and work by the Logic Language Research Group at ICL, Bracknell, into the development of a new logic programming language known as the "Pure Logic Language" [Babb 89a]. These two interests were brought together in a three year Alvey funded project centred at Sheffield City Polytechnic with ICL acting as industrial "uncle".

The research at Sheffield into multiprocessor architectures had been initiated by an interest in data flow programs and the first proposals were for a fine grained non shared memory parallel computer to support this type of application [Loh 82]. This architecture consisted of a fixed two dimensional grid of processing nodes thus providing nearest neighbour connections. Speedy transmission of data through the grid was implemented by having a dedicated message handling processor in each processing node in addition to the actual "working" processor. A simulation of this architecture was produced.

The emphasis in the research moved to considerations of applications in the field of artificial intelligence and in particular the definition of knowledge based systems. Investigation into the possible implementations of the type of semantic networks proposed by Fahlman resulted in a simulation of a network model mapped onto the fixed grid architecture, [Fahlman 79], [Hird 85]. This appeared to be a promising field for development and two projects concerned with parallel systems for knowledge bases were established in 1986. In the first the method for knowledge representation was to be frame based [Brown 87], [Brown 88], [Saeedi 90]; the second project which forms the subject of this thesis was originally defined as "The Implementation of Large Knowledge Bases and Logic Programming Languages on Multiprocessor Architectures" [Jelly 87], [Jelly 88].

On the architectural front the original hope was that the type of fine grained multiprocessor design that had evolved for use with these other applications would prove suitable for implementing a logic language system and that a parallel version of the Pure Logic Language could be mapped to the existing simulation.

The work at ICL on the execution of logic has its origins in database research. The need to define correct and secure database systems which could be extended to include inferencing capacity led to the development of a new logic system [Babb 86a]. This reflected the growing interest in deductive databases in the research community [Gallaire 78], [Gallaire 84], [Minker 88]. Because of problems associated with its operational semantics Prolog was not felt to provide a satisfactory basis for this work and research was initiated to develop a new logic language interpreter which would execute "pure" logic [Babb 86b]. The research at ICL has resulted in the definition of a new language, the Pure Logic Language, and its implementation in the form of an interpreter. Unlike Prolog which is resolution based this interpreter uses a rule rewriting approach: logic expressions are successively transformed by the application of rewrite rules [Nairn 87]. These are of two types: inbuilt system rules and user defined rules, the latter corresponding to the logic program. This is discussed fully in Chapter 4.

Several version of the Pure Logic Language interpreter have been produced by ICL, all based on a sequential mode of operation [Nairn 87], [McBrien 88a], [McBrien 88b]. There has been considerable attention given to parallel logic language systems in recent years and an investigation into the potential for parallel execution of the Pure Logic Language was considered to be important. While work on the sequential system has continued at ICL, the possible parallel execution has been considered in this project and related to other work done in the area of parallel logic languages.

1.2.2. Aims of the Project

The project was set up to bring together work on parallel architectures, knowledge representation and the execution of logic. The hope was that because of its declarative approach to the execution of logic, the Pure Logic Language would prove suitable for defining knowledge based systems. It was recognised that deductive databases and knowledge based systems could be realised by the use of logic programs, and that the programs involved showed some common features. In general they had a comparatively small number of rules and a large number of base predicates. It was one of the aims of the project to consider the implications of this feature for parallel execution of these programs. The intention was to develop a parallel computational model for the Pure Logic Language based on the study of the type of logic programs employed in knowledge based implementations.

The other aim of the project was to consider the design of multiprocessor architectures in the context of parallel logic language execution. It was soon realised that the original fixed grid type of architecture was not ideal for a parallel version of the Pure Logic Language because the topology did not support the type of message passing required, and the idea of mapping a new parallel interpreter onto the original simulation was abandoned. The task became that of identifying the functional requirements for a parallel system and translating them into a new multiprocessor design [Brown 89].

1.2.3. Development and Achievements of the Project

The development of the project is represented diagrammatically in Fig.1.1. It can be seen that the first stage in the project involved a critical appraisal of research in a number of related areas. This led to the definition of an abstract computational model for process based OR parallel execution of the Pure Logic Language. The realisation of the computational model took the form of a parallel language interpreter, and at the same time a detailed design for a new multiprocessor machine was proposed. Finally a simulation of the combined system was produced and quantitative results obtained on the behaviour of the interpreter and the associated architecture.

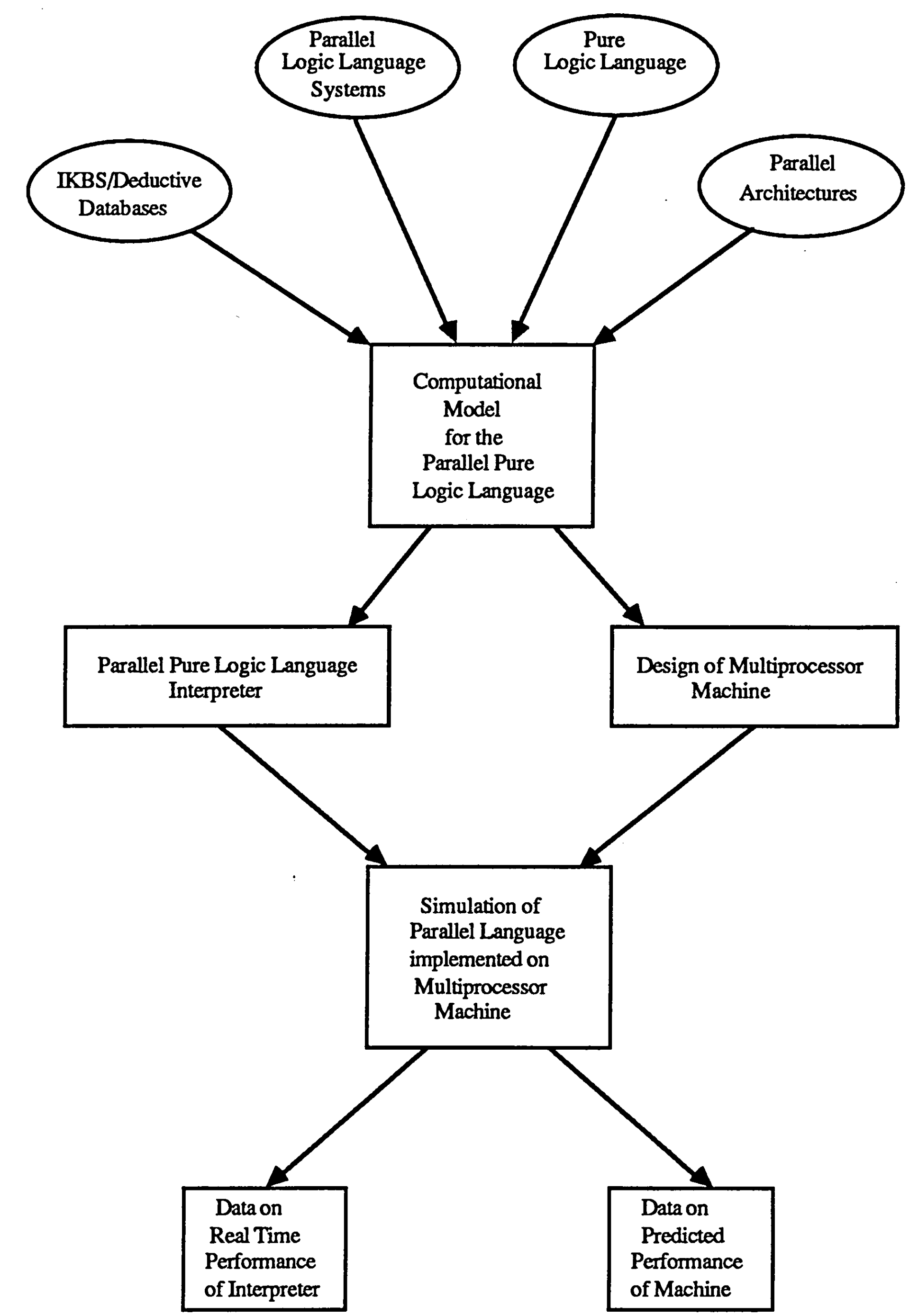


Fig. 1.1 - Project Overview

Four broad areas were covered in the review of related research: the field of knowledge representation including work on deductive databases, parallel logic languages systems, multiprocessor architectures with particular emphasis on those designed for symbolic processing, and the Pure Logic Language itself [Jelly 87], [Jelly 88]. It became clear from the literature review that the parallel execution of logic languages could be implemented in a number of different ways and that the first task in the specification of the system would be to define the type of parallel model required for the Pure Logic Language. The functional requirements for an architecture suitable for the implementation of the parallel system would be dependent on the type of parallelism to be used. In order to define a suitable parallel model for the language, information on the manner in which the sequential system executed was required.

The analysis of the Pure Logic Language involved not only consideration of the theoretical issues involved but a detailed study of the coding of the sequential Pure Logic Language interpreter which had been supplied by ICL. Because the language is based on "pure" logic execution the move to a parallel system had to incorporate a mechanism for the implementation of parallelism without introducing control structures into the language; the underlying interpreter had to be responsible for guaranteeing safe parallel execution.

In general logic languages show the potential for two basic types of parallel execution: these are commonly referred to as AND and OR parallelism [Conery 85]. They arise from the structure of logic programs which incorporate the concept of conjoined and disjoined expressions. In Prolog conjoined expressions are found in the subgoals in the body of a rule definition, and disjunctions arise where there are alternative versions of a rule eg:

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z).

(See Appendix A for the lexical conventions used to represent logic language syntax in this thesis).

AND parallel execution refers to the concurrent evaluation of conjoined subgoals, whereas OR parallelism involves execution of alternative rule definitions and base predicates. Analysis of the Pure Logic Language showed that there was scope for the introduction of both types of parallel behaviour. However the analysis of the execution patterns of the type of program involved in knowledge based systems indicated that only limited performance benefits would be gained by the inclusion of AND parallelism. On the other hand the potential for OR parallel execution within these applications appeared considerable, and the decision was taken to concentrate on this form of parallel activity. The results obtained in the later stages of the project show this belief to be justified.

In this manner the analysis of the relevant research areas led to the proposal for a computational model for the parallel logic language system. This was the first step in the design of a full parallel language system and associated multiprocessor architecture. The computational model proposed allows for the setting up of fully independent OR processes which become candidates for simultaneous execution. The degree to which they are executed in parallel is determined by the characteristics of the multiprocessor machine.

The next objectives in the project involved the specification and coding of a new interpreter to implement the computational model, and the design of an appropriate multiprocessor machine which would match the functional requirements of the parallel language system. The work on the detailed machine proposals was the prime responsibility of John Brown and is documented in [Brown 89], the author of this thesis being responsible for the parallel logic language implementation. The results of this phase of the project were a detailed machine proposal and a parallel interpreter for the Pure Logic Language, and represent a novel approach to the implementation of parallel logic languages.

The specification of the software to implement the OR parallel process model involved work on a new interpreter which would be responsible for the automatic control of these parallel OR processes. It had to incorporate the mechanisms for the creation, execution and transmission of OR processes. Because process creation or spawning had been identified as following a one to many pattern, ie one parent process spawned several offspring processes, the interpreter was responsible for the handling of groups of processes each time a disjoined expression was encountered. Because the computational model had been based on the notion of fully independent processes, each newly created process had to incorporate all the information required for it to complete its execution without reference to the parent, and this information had to be transmitted from parent to offspring at the time of process spawning. The new parallel interpreter retained the rewrite rule approach defined in the sequential version but the move to a process based OR system involved the production of a new rewrite rule module redefining the inbuilt system rules to incorporate the mechanism to implement OR parallel execution.

At the same time as the new parallel interpreter was being written work was carried out on the functional requirements for a suitable architecture on which to implement the parallel language system and a new machine design was prepared [Brown 89]. This design incorporated a method of implementing concurrent broadcasting operations which matched the one to many pattern of communication employed in the parallel interpreter. The incorporation of this form of broadcast communication in the parallel language system and its direct mapping onto the proposed machine design represent the project's main theoretical contribution to this field of research. It is believed that this approach to the implementation of a parallel logic language system is new and offers an effective mechanism for communicating information between separate processes operating on different processing elements.

The final stage in the project involved the design and implementation of a software simulation of the broadcast multiprocessor machine. This was interfaced with the parallel interpreter to provide a system which mapped the parallel logic language onto the architecture. Measurements about the predicted performance of the system were made: these involved data about the behaviour of the interpreter as well as information on the operation of the multiprocessor machine. It can be seen from these results that the system is able to utilise the potential OR parallelism within the programs to give considerable performance benefits. The results form the basis of the detailed evaluation of the system and proposals for future work in this area.

1.3. Organisation of the Thesis

The structure of the thesis is closely related to the chronological development of the project. Chapters 2 and 3 document the important aspects of the background work that were considered in the first phase of the project. This is followed in Chapter 4 by a detailed consideration of the Pure Logic Language and the method by which the sequential system is implemented. Comparisons are made between the mode of execution of the Pure Logic Language and Prolog.

The new parallel version of the Pure Logic Language is described in Chapter 5. This shows the theoretical considerations in the move to a computational model for OR parallel process execution, as well as the implementational details. The functional requirements of the multiprocessor system designed for the parallel language are discussed in Chapter 6 and proposals for a hardware realisation presented.

The two aspects of the project, ie the new parallel interpreter and the multiprocessor design, are incorporated in a simulation of the proposed system. This is described in Chapter 7. Chapters 8 and 9 are concerned with the testing of the simulation and the analysis of the results obtained. Chapter 10 presents an evaluation of the project and indicates the areas into which future work could be directed.

1.4. Summary

This chapter introduces the work in the thesis. The research described in it is in the field of parallel implementation of logic languages with particular emphasis on knowledge based systems applications. The work leading to the inauguration of the project has been discussed and its aims and development have been outlined. The project has contributed to the body of knowledge in this area by the proposals for a new parallel logic systems and an associated multiprocessor architecture.

Chapter Two

Parallelism in Knowledge Based Systems and Logic Languages

2.1. Introduction

The aim of this chapter and the next chapter is to set the scene for the work done during this project. There are two main areas which have been brought together in the work: the field of knowledge representation and manipulation with particular emphasis on the use of logic programming languages, and the design of multiprocessor machines performing parallel computations. It has been shown in Chapter 1 that the project has focused on the employment of a new logic programming language, the Pure Logic Language, as a suitable knowledge representation formalism, and has developed a parallel system based on its use. This chapter documents the process of narrowing down the area of interest from generalised knowledge based systems and their implementations to considerations for the design of parallel logic language systems. Specific examples of parallel logic language systems and associated architectures are discussed in Chapter 3 and the Pure Logic Language will be considered in Chapter 4.

The chapter looks at the concept of knowledge based systems and briefly at the different types of knowledge representation that can be used in these systems, drawing on the corresponding database experience where appropriate. This is followed by a discussion on the inclusion of inferencing capacity within such systems, and the use of logic programming languages as the unifying formalism for rules and data is presented. The concepts of deductive databases and Datalog programs are introduced at this stage.

When the implementation of a knowledge based system is considered, it is recognised that the inclusion of deductive capacity involves high computational demands and this has led to many proposals for parallel execution for such systems. Parallel execution can take the form of specialised hardware to tackle one particular task within a conventional sequential system, or the development of an integrated computational model based on parallel execution. It is the latter group of systems that are important to this project and particularly those parallel execution models that involve logic programming languages. The potential for parallelism in logic languages will be discussed; this has been an active research area over the past ten years and many proposals have been put forward.

The next chapter will review examples of parallel logic systems and consider the implications for the design of multiprocessor machines for their implementation.

2.2. Knowledge Based Systems

2.2.1. Introduction

The terms "intelligent knowledge based systems" (IKBS) or "knowledge bases" are increasingly used not only in the research community but in the commercial world. The type of applications in which knowledge bases are used include expert systems, natural language processing, deductive databases and other systems which incorporate inferencing mechanisms [Frost 86]. The interest in this type of system has developed into a major research area within the field of artificial intelligence; at the same time work in extending conventional databases to include deductive capacity has addressed the same issues [Gallaire 78], [Gallaire 84], [Minker 88], [Gardarin 89].

Definitions of the term "knowledge base" vary from author to author but it is generally agreed that a knowledge based system will contain an inferencing mechanism as well as data, ie it is the application of an "intelligent reasoning mechanism to an explicit representation of knowledge" [Hogger 84]. At its fundamental level therefore a knowledge base is a "collection of simple facts and general rules representing some universe of discourse" [Frost 86]. The term "data" is used to represent the "collection of simple facts" and thus a "data" base plus general rules becomes a "knowledge" base. The concept of "information" is important: this has a "value added" connotation: knowledge becomes information when it tells the user something he or she did not already know, or in information theory parlance "reduces the receiver's uncertainty about some aspect of the universe of discourse" [Frost 86]. It is important to realise that it is "information" in this sense that the user of a knowledge based system requires. The knowledge that it is snowing heavily in the French Alps is not likely to be of great benefit to the people of Chamonix but may be important information for someone in Sheffield planning a skiing holiday.

If a knowledge based system is considered in this manner it can be seen that the implementation of the system has two aspects: the choice of an appropriate knowledge representation formalism and the inclusion of an inferencing mechanism. However this notion of "rules plus data" presents a structuring problem, ie to what extent should the knowledge representation model impose a predefined structure on data to be encapsulated. For some applications a structured approach provides immediate advantages, allowing relationships to be expressed naturally and enabling communication between users of the system to take place easily. For other types of system a less structured approach allows different types of relationship to be expressed without the necessity to mould data into unsuitable formats.

The main types of structured models used in the artificial intelligence field are often referred to as "slot and filler" knowledge representations: these include semantic nets, frames, scripts, conceptual dependencies and structures [Brachman 85], [Fahlman 79], [Minsky 74], [Minsky 85], [Schank 75], [Schank 77], [Sowa 84], [Woods 85]. Work on these formalisms was started in the 1970s and has resulted in a considerable research literature as well as a number of commercial systems, eg KEE [KEE 86]. This work has direct parallels with research from the software engineering field into the theory of data typing, and it is interesting to note that the current work on object orientated programming shows a marked similarity to those concepts developed for frame based systems [Meyer 88]. Recently proposed object orientated database systems are also incorporating the concepts of hierarchical organisation and inheritance of attributes that are familiar from the earlier artificial intelligence work on frames [McGregor 90], [Gray 90a].

Less structured knowledge representation models can be seen in the relational database approach where normalised relations are stored in tables of tuples, and there are no explicit links between relations [Codd 71]. A similar approach is taken in logic programming languages where data is stored in sets of base predicates ("facts" in Prolog) which are equivalent to relations [Gray 84], [Hogger 84]. The links between the base predicates or relations have to be made explicitly in the form of rules, unlike the frame based approach where meta rules covering aspects such as inheritance of attributes are implicitly built into the structure of the system. Fig.2.1 gives an example of a small relational database which consists of two relations, and the Prolog program that includes the equivalent base predicates and a rule.

Relational Database

Relation 1

TName	Course
Dr Good	CS
Ms Sharp	IT
Prof Wright	Maths
Mr Smart	Maths

Relation 2

SName	Course
Sally White	IT
Bill Gray	CS
Sue Brown	Maths
Frank Green	CS

Prolog

```
teaches(X,Y) :-
        teacher(X,Course),
        student(Y,Course).

teacher(drgood,cs).
teacher(mssharp,it).
teacher(profwright,maths).
teacher(mrsmart,maths).

student(sallywhite,it).
student(billgray,cs).
student(suebrown,maths).
student(frankgreen,cs).
```

Fig. 2.1 - Relational Database and Prolog Program

The factor that promotes the relational database model into that for a knowledge base has been defined as the incorporation of general rules [Gardarin 89]. This involves the inclusion of a reasoning mechanism for the manipulation of the rules and data to produce data in a form that is not explicitly present. This inferencing capacity is at the heart of knowledge based systems; most of these systems will include an automated deductive system, but other forms of reasoning, eg abduction and analogical reasoning, are used in some applications [Frost 86]. It is outside the scope of this project to consider the role of these other inferencing mechanisms: the concern here is with deductive capacity. The following sections on logic programming and deductive databases will indicate how classical deductive methods can be implemented in an automated system.

2.2.2. Logic Programming

Logic programming has evolved from theoretical work on automated theorem proving [Lloyd 84]. The previous section has referred to the use of predicate logic clauses as a means of knowledge representation. This model meets the basic requirements for a knowledge representation formalism in that it provides an unambiguous interpretation, allows the system to be reasoned about and facilitates communication between users. The problem in the use of predicate logic as an executable language is that application of a deductive method is liable to produce an extremely large number of deductions which although valid are not useful to the user; in other words the search space is uncontrollably large. The work by Robinson defining the resolution principle provided the means of controlling this situation and made possible the efficient automation of deduction [Robinson 65]. By restricting the knowledge representation language to Horn clauses, an interpreter could be produced which was not only sound but computationally efficient. The work of Colmerauer and Kowalski led to the development of the logic programming language Prolog [Colmerauer 73] [Kowalski 74]. This language is now widely used and familiarity with it is assumed [Clocksin 81].

The crucial feature that emerged from Kowalski's work was that clauses written in Horn clause logic have both a declarative and a procedural interpretation [Kowalski 74]. The declarative interpretation of a logic program rests with its definition as a clause set and specifies the relationship that exists between the head (left hand side) and body (right hand side) of the rule. The same logic program can be given a procedural interpretation which defines the operational semantics of the language. A clause can be regarded as a program and subclauses as procedures. Program execution involves the calling of appropriate procedures for each subgoal. Thus resolution can be defined in algorithmic terms using the procedural semantics of a conventional programming language. If the query
(contains europe, X). is put to the following program or clause set:

```
partof(europe, britain).
partof(britain, london).
partof(europe, france).
partof(france, paris).
contains(X,X).
contains(X,Z) :- partof(X,Y), contains(Y,Z).
```

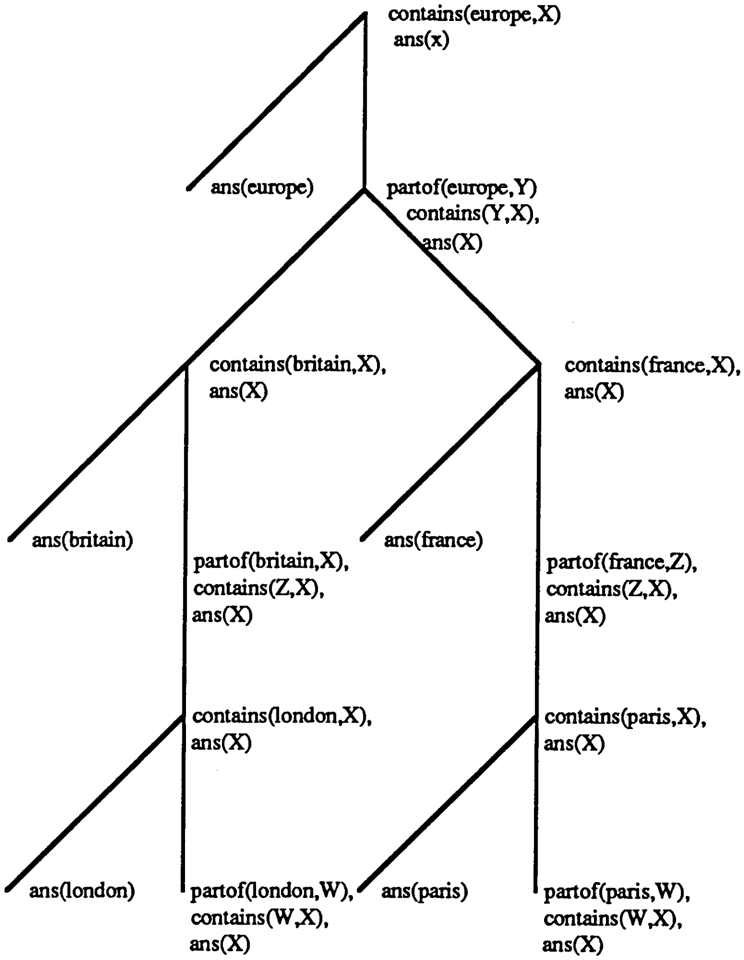


Fig. 2.2 - Resolution Search Tree

the resulting resolution search tree is shown in Fig.2.2 (adapted from [Warren 88b]). Branching in the search tree occurs when alternative predicate definitions are present. Each node in the resolution search tree can be regarded as defining a procedure calling operation: the appropriate subgoal (in the case of Prolog the left hand one) is selected as the next call, the procedure whose name matches the call is invoked and the formal and actual parameters are unified. The body of this new procedure replaces the call in the goal list with appropriate unifiers applied, thus producing a new goal list. This method of handling literals in a clause as procedure calls allows a logic language program to be executed in a similar manner to a conventional imperative program. Where no alternative definitions of predicates exist the flow of computation is directly comparable to that produced in an imperative language.

However alternatives within logic languages have to be handled in a different manner because it may be necessary to "backtrack" to a previous state of the computation. This is known as "non determinism": at a general level Hogger defines a non deterministic program as one which "admits more than one computation, that is, has a branched computation tree" [Hogger 84]. However this feature of non determinism in logic languages is somewhat different from the situation that exists in conventional procedural languages. Although the flow of computation in an imperative language can exhibit branch or choice points, eg

```
if (condition)
      {do action1 }
else
      {do action2 }
```

the branch representing the unsatisfied condition is always discarded, and there is never a need to maintain information about the computational state of an branch which has not been selected. In logic languages backtracking to explore previously marked choice points is the method by which the search tree is explored and the implementation of logic languages has to involve the storage of information relating to these branching points. This is discussed in Chapter 2.2.6 in relation to the implementation of Prolog.

It is the procedural interpretation that allows an automated system to be written to execute the language, ie to make the refutation proof [Lloyd 84]. The methods by which the execution of logic language systems is implemented are considered in Chapter 2.2.6. It is important to note that the use of a logic programming language allows rules and data to be represented in the same formalism and the inferencing mechanism handles both aspects in a uniform manner. This makes the use of logic programming languages for representing knowledge attractive.

2.2.3. Deductive Databases

At the same time as work on logic programming languages was developing out of research on computational logic, the application of logic to the database field was being considered. Work by Reiter, Chang and others on the relational database model originally proposed by Codd, put a logical interpretation on the model and introduced concepts such as the Closed World Assumption in order to allow negation to be handled correctly in the system [Reiter 78a], [Reiter 78b], [Chang 78]. The concept of extending the relational database to include general rules results in the definition of a deductive database. The foundations of the research into deductive databases have been reviewed by Gallaire and Minker [Gallaire 78], [Gallaire 84], [Minker 88].

From a formal viewpoint a database can be viewed in two ways; the model theoretic and the proof theoretic viewpoint [Gallaire 84], [Gardarin 89]. From the simpler viewpoint the relational database is considered to be a model of a first order logic. Thus a predicate name in a first order logic formula corresponds to a relation name. The values in the database are the set of constants satisfying the formulae and queries are treated as expressions whose truth value can be ascertained with respect to the database. However this view does not allow for inferencing techniques to be included into relational database theory and therefore a second approach to the database is defined as the "proof theoretic". In this view the database is seen as a set of logic formulae that can be used for inferring new formulae, ie as a set of axioms of a first order logic. In the proof theoretic approach the theory requires additional general axioms to be included concerning domain closure, completeness, unique names etc. Having defined the database in terms of the general and specific axioms, a general proof mechanism, such as resolution, can be used; this provides the formal interpretation of a deductive database.

The design of deductive databases involve issues such as how to implement the inferencing mechanism efficiently, to what degree the rule handling element should be incorporated within the database management system, and the need for common or separate languages to handle rule definition and data manipulation. The question of language for these tasks is important: relational databases use some variant of relational algebra (or its declarative counterpart, relational calculus) as the method of expressing the operations for the system. Relational algebra defines the set operations which can be applied to the relations in the database in order to derive new relations. A detailed description of these operations is given in "Introduction to Database Systems" [Date 85]. It is worth considering an example of the database Join operation as this is referred to in the section on parallel logic languages. The relations in Fig.2.3 show the small previously defined database holding information on students and teachers (see Chapter 2.2.1). Relation3 which was not originally in the database has been derived

Relation 1

TName	Course
Dr Good	CS
Ms Sharp	IT
Prof Wright	Maths
Mr Smart	Maths

Relation 2

SName	Course
Sally White	IT
Bill Gray	CS
Sue Brown	Maths
Frank Green	CS

Relation 3

TName	SName	Course
Dr Good	Bill Gray	CS
Dr Good	Frank Green	CS
Ms Sharp	Sally White	IT
Prof Wright	Sue Brown	Maths
Mr Smart	Sue Brown	Maths

Fig. 2.3 - Relational Database with Derived Relation

by performing a Join operation, ie by combining Relation1 and Relation2 with respect to the value of their common attribute "course". This is expressed as

Relation1 >< Relation2
or
JOIN(Relation1, Relation2).

However languages (such as SQL) derived from relational algebra or calculus cannot handle the inclusion of rules which are needed in a deductive database system, and thus designers of these systems have looked towards the logic languages as providing suitable rule definition and query answering facilities [Maher 88]. The concept of Datalog programs and their use in deductive databases is looked at in the next section (Chapter 2.2.4).

The implementation of a deductive database system can take a number of different approaches. The degree to which the inferencing system is integrated with data management can vary from "loose coupling" to "full integration" [Gardarin 89]. A fully integrated system implies that there is no separation between the inferencing mechanism and the data storage management; they are integrated together at a low level of system implementation. The actual database management system is designed in such a manner as to include the rule base management and the logic interpreter to handle queries; the languages which handle rule and data manipulation and querying are likely to use a common syntax, and the user is presented with an integrated interface to the system. "Coupled" systems use a conventional database management system as the underlying storage and manipulation mechanism, and the inferencing component is "bolted" onto this. In a loosely coupled system the inferencing mechanism is likely to represent rules in a different language from that used to define and handle the data, whereas tighter coupling implies that rules and data are expressed in the same formalism, and the storage management is hidden from the user. Fig.2.4 which has been adapted from [Gardarin 98] gives a schematic representation of these three approaches.

An example of the tightly coupled type of system is the Prolog/PSAlgol system where rules and data are expressed in Prolog but the underlying storage mechanism for the Prolog "facts" is a database management system [Moffat 86], [Gray 87a]. Work on a Prolog database system at Edinburgh is concentrating on the organisation of Prolog modules and files for disk storage in order to implement a fully integrated deductive database system [Williams 87a].

However because of its "non" logical features Prolog does not provide a formal model for a system definition and manipulation language for deductive databases, and this has led to the concept of Datalog programs. These are discussed in the next section.

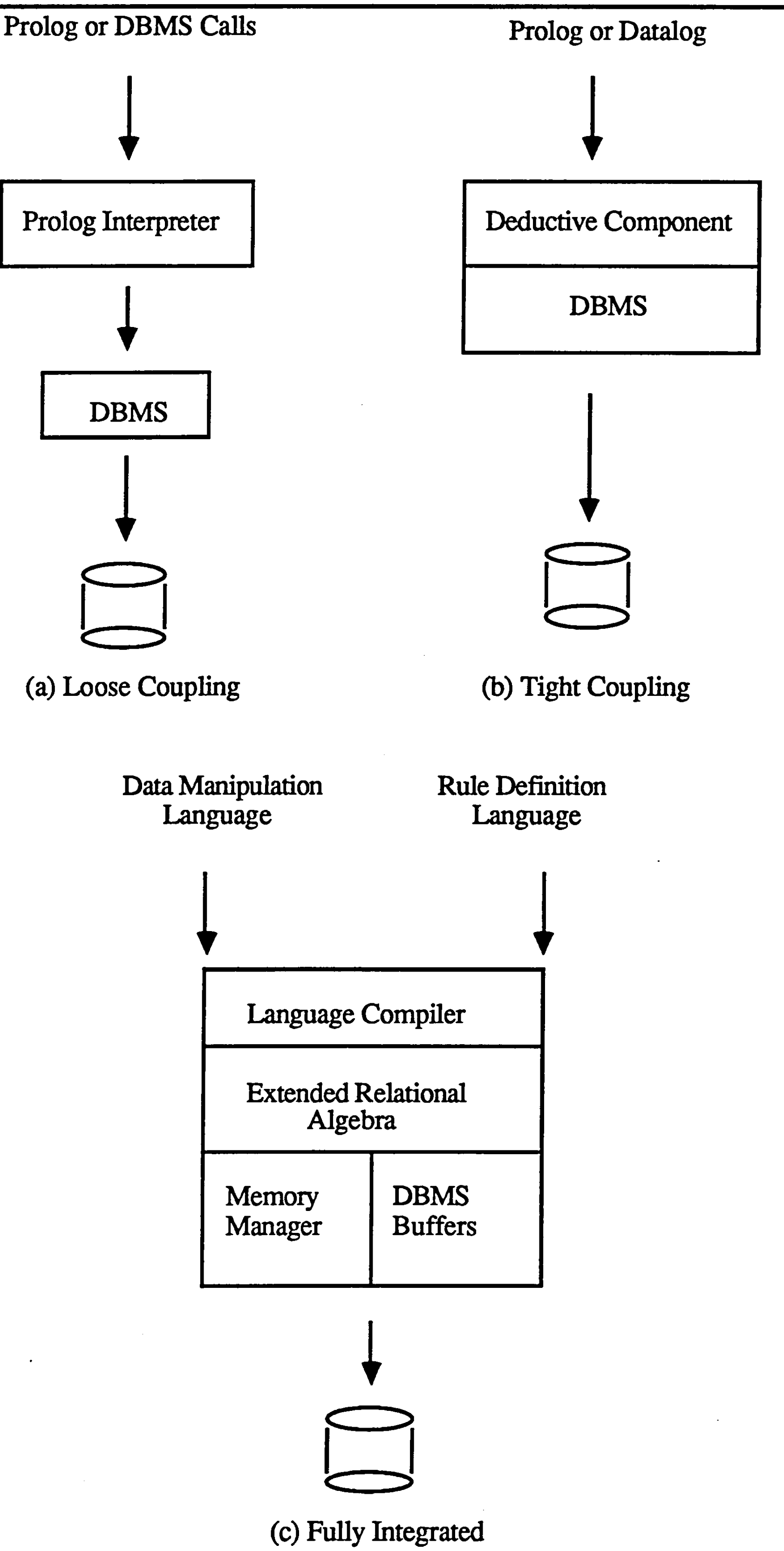


Fig. 2.4 - Deductive Database Implementation

2.2.4. Datalog Programs

Datalog programs can be used to provide a formal system definition and a query language: they specify the database in terms of Horn clauses without functions symbols, ie they can be regarded as Horn clause logic programs with no extra logical features, functions or negation [Minker 88]. Thus the semantics for a Datalog program can be regarded as having a declarative or procedural interpretation in the same manner as generalised logic programming [Kale 88a].

Several extensions to the concept of Datalog programs have been put forward in order to enhance their usefulness as a database definition formalism. These include the incorporation of negation: negated predicates are allowed in the rule body. In order to allow a unique "least model", a program which includes negation has to be stratified; the program is divided up into levels or strata, and predicates can only be negated if they have been fully defined in one of the previous strata [Gardarin 89]. Similar extensions may be provided for the inclusion of functions and set operations.

Datalog programs can be defined using a specific syntax for a particular database system, or by employing a subset of a logic language such as Prolog or the Pure Logic Language. They allow the system designer to express both the intensional and extensional database in the same language, the complete program defining a "logic database" [Gardarin 89]. The first benchmark program with its associated queries given in Appendix C is an example of a Datalog program written in the Pure Logic Language.

2.2.5. Implementation of Logic Based Knowledge Bases

As has been seen the concept of a knowledge based system includes not only the storage of data but a reasoning component and thus the question of efficient implementation of such a system has to address both aspects. As this project is primarily concerned with the use of logic languages as a knowledge representation method the discussion to follow will concentrate on systems using this type of model.

In general the inclusion of an inferencing mechanism puts heavy computational demands on a system because the algorithms used to implement it define the testing of many different hypothesis. These tests usually involve a series of pattern matching operations, the results of which are discarded if the match fails. In this way systems with deductive capacity generate a search space for each query and the effective management of the search process determines the performance of the system [Charniak 85]. It has been shown that the search space for logic programs can be reduced by the application of theoretical concepts such as an SLD resolution based refutation proof (see Fig.2.2 for an example of a small resolution search tree). Helpful as this is, for a logic based program which includes a considerable number of alternative definitions, query response is still going to involve computational operations whose results do not contribute directly to the answer.

The question of implementing efficient methods for handling the heavy computational demands of these systems can be tackled in three general ways:
a) the use of parallel hardware to allow the simultaneous execution of different computations,
b) the development of separate methods for implementing the inferencing component and the data management aspect,
c) the introduction of special compiler techniques to produce efficient "tailor made" sequential code for each inferencing procedure in a given program. This is looked at in more detail in Chapter 2.2.6.

It is important to note that these three approaches are not mutually exclusive and many systems based on parallel architectures involve aspects from categories b) and c). The question of parallelism in systems based on logic languages is looked at in detail in Chapter 2.3 and Chapter 3.

The separation of rules and data handling has been referred to in the section on deductive databases. This approach allows the designer of the system to utilise the considerable knowledge of data management gained in the conventional database field. Because of the scale of the systems involved, many deductive database proposals have concentrated on using inferencing methods which avoid the need to implement a full resolution proof. This can be done by separating the "intensional", ie rule handling, aspects of the system from the "extensional", ie data management. This allows methods of query optimisation to be introduced in order to ensure that access to the data is kept to a minimum. The inclusion of recursive rules adds complexity to this process and several proposals for query transformations have been made in order to cut down the overheads involved in recursion [Bancilhon 86], [Valduriez 86], [Ramakrishnan 88].

The question of data handling involves not only the conceptual level aspects of representation and relationships but inevitably the method of disk storage and retrieval. Indexing schemes such as that proposed for deductive databases by Lloyd [Lloyd 81] are included here. Another approach is the design of specialised hardware which aims to give rapid access to appropriate disk stored data: this includes systems such as CAFS [Howarth 85]. In order to meet the disk retrieval demands of knowledge based systems there has been considerable work on a number of different hardware systems which can be regarded as "backend" machines offering fast associative access to data. The paper by Gray gives a summary of work on these systems in Britain [Gray 87b]. It is outside the scope of this project to give detailed consideration to methods, which may involve hardware and/or indexing schemes, for handling secondary storage data, but it is recognised that this area is of great importance in the implementation of realistically large knowledge based systems.

2.2.6. Implementation of Prolog

In Chapter 2.2.2 an outline description of the procedural interpretation of resolution based logic languages such as Prolog was given. This section looks at this in more detail and specifically how the system can implemented in a compiled version.

There are two components which implement the procedural semantics for a logic language such as Prolog: first the choice of which procedure is to be the next candidate for execution is determined by the "call selection" or "computation" rule. The second component of the interpreter is a matching or "unification" procedure which is invoked each time a goal or subgoal is selected. The unification procedure is responsible for determining whether a call to a subgoal succeeds or fails, and in the case of success may produce binding values for uninstantiated variables [Lloyd 84].

The standard computation rule used by Prolog to implement the resolution proof always selects the first call in the goal, replacing it with the body of the new procedure [Hogger 84]; this leads to a depth first exploration of the search tree. When alternatives are encountered and branching occurs, the branch points are stored by the system as backtrack points, ie the place at which the computation must resume in the event of failure of a branch or if a full set of bindings is required.

From this brief description of the execution process in a Prolog program it can be seen that the internal state of the computation at any given time can be represented by the current position relative to the search tree. Branches that have been explored can be discarded, those that have still to be explored must be stored as choice or backtrack points, and the present state of the computation or the "environment" is represented by a goal list plus any current variable bindings. The ability to return to a backtrack point and pick up the computation from that point can only occur if the environment that existed at that point is also stored. Thus the information required for backtracking involves the storage of previous bindings and goal lists.

An interpreted Prolog system includes data structures to represent the original clauses or program, the present state of the computation including current variable bindings and information on previous environments to allow backtracking to occur where appropriate. The execution of the program will be driven by generalised algorithms implementing the call selection or computation rule, and the unification operation.

However the use of these generalised algorithms applied at run time will often involve unnecessary computations. Most recent implementations of Prolog have abandoned the use of general computation and unification algorithms in favour of a compiled system. At program insertion time each procedure, ie each group of clauses defining a predicate, is compiled into low level code which specialises the unification and computation rules for that procedure. This low level code replaces the original program and there is no need for the general interpreter code to be held in the system. At run time the compiled code which has specialised the deductive process for each goal call is used at run time and results in substantial savings in terms of unnecessary computations.

The work by Warren in producing a compiler which translates Prolog to an abstract machine code has provided the basis for most current commercial Prolog systems [Warren 88b]. The Warren Abstract Machine (WAM) also defines the memory management of the system with the use of

```
Prolog Rule:

concatenate([], L, L).
concatentate([X|L1], L2, [X|L3]) :- concatenate(L1, L2, L3).


WAM Code for rule:

concatenate/3:
          switch_on_term C1a, C1, C2, fail.

C1a :     try_me_else C2a              % concatenate(
C1  :     get_constant nil, A1         %    [],
          get value A2, A3             %    L, L
          proceed                      % ).

C2a :     trust_me                     % concatenate(
C2  :     get_list A1                  %    [
          unify_variable X4            %       X|
          unify_variable A1            %          L1], L2,
          get _list A3                 %    [
          unify_value X4               %       X|
          unify_variable A3            %          L3]) :-
          execute concatentate/3       % concatenate(L1, L2, L3).
```

Fig. 2.5 - Compiled Prolog Code

various stacks and registers to hold control data such as backtrack or choice points. The performance benefits are further enhanced by the incorporation of indexing methods for access to base predicates. The use of the Warren Abstract Machine has also been employed in many parallel Prolog systems and it has set the standard for the implementation of high performance sequential Prolog systems such as Quintus Prolog and BIM Prolog .

Fig. 2.5 shows the WAM instructions produced by the Prolog compiler for the rule for list concatenation: these are included as an example of the intermediate code, ie the generalised machine instructions which can then be translated into the native machine code of the target machine [Warren 88b]. Programs can either be stored as WAM (ie intermediate) code or specific machine code.

It is worth looking in outline at the WAM data structures that are used to control the program execution. Many parallel logic programming languages are based on these data structures and the implementation of the Pure Logic Language which is discussed in Chapter 4 shows some similarities.

The WAM data structures which hold the information on the computation state are given below in Fig.2.6. The code area holds the compiled code for each rule in the program, the "heap" is used to store large structures (typically lists) constructed during execution, the "trail" holds conditionally bound variables and the "push-down list" is a small stack used during unification. The registers are shown in Fig.2.7. For the purpose of this description the memory area of prime importance is the "local stack" which contains the data on the state of the computation throughout execution. Details of the role of the other areas can be found in [Warren 88b], [Maier 88].

Two types of data structures are stored on the local stack. It has been shown in Chapter 2.2.2 that the system must hold information on the current "environment", ie the current goal list plus bound variable values, and information to enable backtracking to occur. The current environment of the execution is given by a chain of "environment frames" which are stored on the stack. Each time a call is made to a new subgoal a frame is created and linked to its parent, ie the previous subgoal, by a pointer storing the parental frame address (the CE or continuation environment pointer). The address of the code for the next call to be made after the present subgoal is evaluated is also stored in the environment frame (the CP or continuation code pointer). Thus the chain of frames created by the CE and CP pointer links indicate the present state of the goal list. Also included in the environment frames are details of bound variables. The frame may hold the actual binding value or the pointer to it in the case of structured terms. In the execution of a program which contains no alternatives the state of the computation is defined by this chain of environment frames.

However any implementation of Prolog has to handle backtracking when alternatives are encountered. In order to allow backtracking to occur the system has to store the state of the computation at the time of branching so that it can be re-established on backtracking. This is done by the use of a choice point frame which is also added to the local stack. In a manner similar to the chaining of environment frames a chained list of choice point frames is maintained representing backtrack points in reverse chronological order, ie the most recent choice point is at the head of the chain and its position is given by the register B holding its address. Each choice point

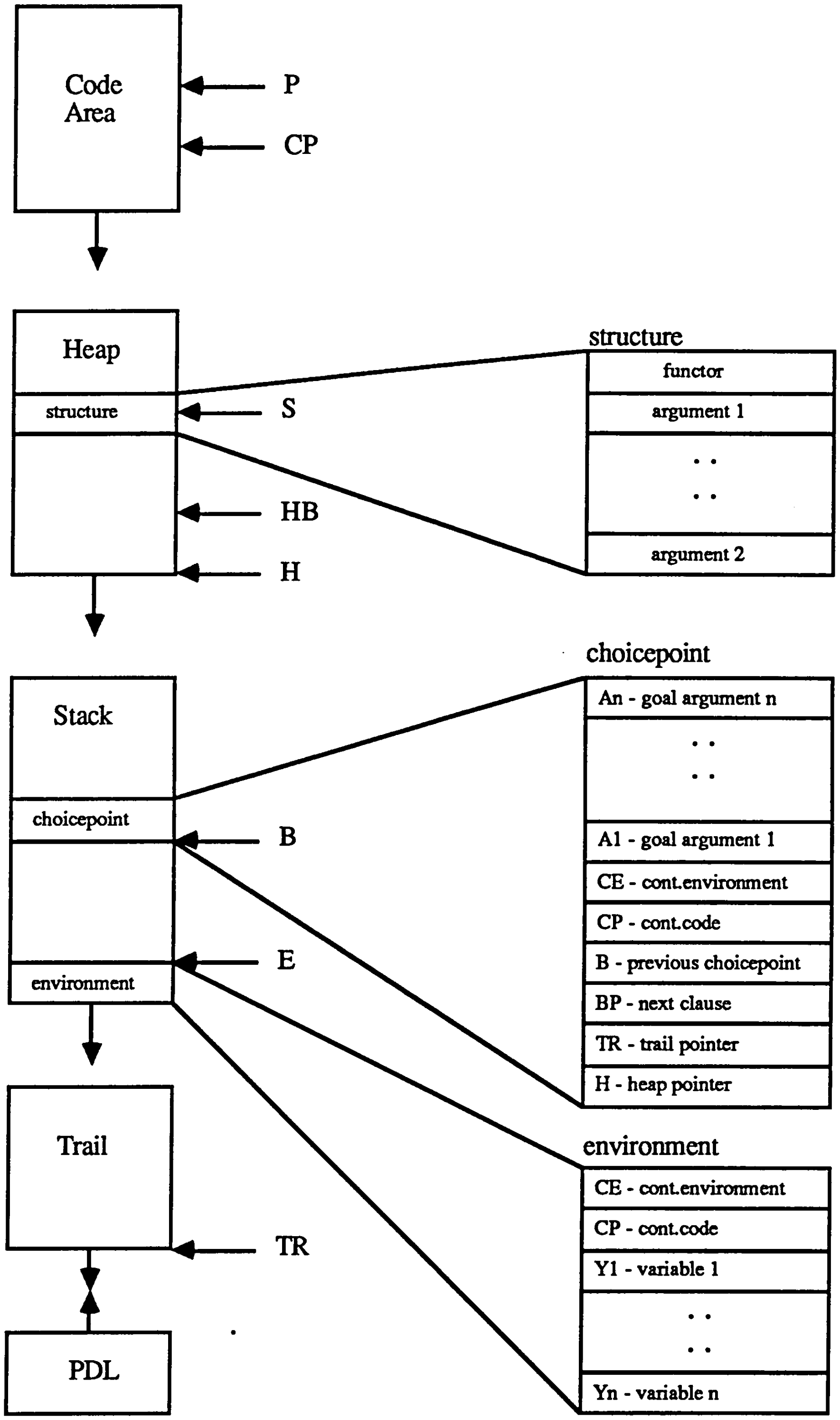


Fig. 2.6 - WAM Memory Organisation

frame stores copies of all the registers of the WAM at the time of its creation thus enabling the computational state to be restored - see Fig.2.7.

P	program pointer	code area
CP	continuation program counter	code area
E	last environment	stack
B	last choicepoint	stack
TR	top of trail	trail
H	top of heap	heap
HB	heap backtrack point	heap
S	structure pointer	heap
A1, A2, ...	argument registers	
X1, X2, ...	temporary variables	

Fig. 2.7. - WAM Registers

In this manner the run time or local stack consists of a mixture of choice point and environment frames, the currently executing call represented by the frame at the top of the stack. The performance of the WAM is improved by a number of optimisations which aim at saving space and unnecessary frame creation but it is not appropriate to consider this aspect of Prolog implementation here. This outline description of the standard form of compiled Prolog is sufficient to allow comparison between it and the Pure Logic Language interpreter to be made in Chapter 4.7.

2.3. Parallelism in Logic Languages

2.3.1. Introduction

This section looks at the case for parallel execution of logic languages and it will be followed in Chapter 3 by a description of examples of such systems. Parallelism in logic languages covers a wide field and involves theoretical concepts and implementational problems. Work on parallel logic languages systems has been produced by a number of large centres throughout the world and it is still an active research area. Considerable difficulty has been experienced in moving from a simple conceptual model of parallel execution to real implementations, and the reasons for this will be discussed. The abstract level of defining parallelism in logic languages is seductive in its simplicity but methods of implementing it lead to a morass of problems.

The section includes a discussion on the different concepts of parallelism and concurrency, and looks at issues of the control of parallel behaviour in systems based on these two abstractions. Logic language programs exhibit the potential for a number of different types of parallel execution [Conery 83], [Conery 85], [Hogger 84] and these are discussed with particular emphasis placed on the fundamental concepts of AND and OR parallelism. The implementational problems of employing these forms of parallelism are looked at, and it is shown that AND parallel execution involves computational overheads not involved in OR parallel systems. The potential performance benefits to be gained from parallel execution are considered and the conclusion reached that OR parallel execution is likely to provide substantial speedups for the type of application that this project is concerned with. The usefulness of AND parallel execution is more in doubt.

2.3.2. Concurrency and Parallelism

These terms are both used in the work on logic programming languages, and it is necessary to consider their precise meaning. The concept of concurrency is known from the work by Dijkstra and Hoare and involves the defining of computational modules which can be executed simultaneously in safety; concurrent programming languages present a method of representing this computational possibility and expressing the communication between the concurrent processes, and use methods for the explicitly expressed control of concurrency [Dijkstra 68], [Hoare 78]. Because of the involvement of communication, it is perfectly possible to define two concurrent processes which will necessarily be executed serially because of the nature of the communication between them. The traditional producer-consumer process model is an example of this: the fact that a consumer process can only execute following a producer process does not invalidate the description of them as concurrent processes. Thus concurrent programming is a paradigm for expressing relationships between different parts of the computational task: its primary aim is to produce a valid model for the problem under consideration, performance benefits due to simultaneous execution are a secondary issue.

Parallelism on the other hand can be seen as the search for performance benefits by dividing the computational task into many simultaneous operations. This would include specialised operations such as vector processing, low level pipelining of instructions in CPUs as well as the coarse grained parallel execution of concurrent processes. The goal of producing a parallel model for logic programming language execution is to achieve speedups in performance, whereas the definition of a concurrent logic programming language involves other aims and is likely to be directed to different types of applications. However the potential for defining a parallel or concurrent model is based on the concepts of conjunction and disjunction within logic languages and this is discussed in Chapter 2.3.4.

2.3.3. Control of Parallelism

Linked to the question of language model in parallel systems is the issue of explicit control. In a concurrent language it is an implied aspect of the language that the concurrent behaviour is specified, ie the programmer uses algorithms to define processes which are candidates for simultaneous execution. The system may not implement them simultaneously owing to constraints on resources, but the programmer has written the program with specific annotations to indicate concurrent control. This is equally true of procedural concurrent languages such as Occam and of logic programming languages such as Concurrent Prolog, Parlog and Guarded Horn Clauses [INMOS 88], [Shapiro 83], [Shapiro 86], [Clark 83], [Clark 86], [Ueda 86].

The situation with languages used to implement the parallel model is slightly different. In these systems it is not a necessary condition that they should explicitly represent simultaneous operations; the parallel behaviour in the system can be determined either by automatically invoked operating system type procedures, or by the programmer's use of specific annotations within the language. Examples of both types of system will be discussed in the section on parallel logic languages.

The question of automatic versus programmer control of parallelism is still an active research issue. The advantages of automated parallelism are clear: the absence of control annotations makes program debugging and verification easier and it allows programs already written in a standard sequential language such as Prolog to be run on a parallel machine without alteration [Butler 88]. Examples of the type of control annotations used in parallel logic language systems are given in Chapter 3.1.2.3. The addition of these program annotations contradicts the declarative concept of a logic language. (This is equally true of the commonly used program construct for the control of backtracking in sequential execution, ie the cut). However many systems do include such control structures: first because the programmer can use his or her knowledge of the program and the hardware to ensure that the right amount of parallel execution takes place; secondly to avoid some of the pitfalls caused by theoretical issues in parallel logic languages. These theoretical problems which involve the role of the shared variable in conjoined subexpressions, are looked at in Chapter 2.3.4.4. For both these reasons many systems have incorporated the notion of programmer control of parallel execution as a necessary part of the program. In this project the aim has been to maintain the purely declarative nature of the Pure Logic Language and thus parallel behaviour is generated automatically by the underlying system. This will be discussed more fully in the following chapters but the price that has to be paid is that the system may lack some of the fine tuning that other parallel logic language systems can achieve.

2.3.4. Sources of Parallelism

2.3.4.1. Introduction

Reference is commonly made to four sources for potential parallelism within logic languages: AND, OR, search and stream parallelism [Conery 83], [Conery 85], [Hogger 84].

2.3.4.2. Search Parallelism

Search parallelism is different from the three other types in that it is less closely related to the logic programming model. It refers to the process of finding clauses from the program that match a given expression; instead of running through the clauses in textual order attempting a match the system initiates a parallel or associative search. As such the implementation of search parallelism would appear highly advantageous; however most efficient Prolog systems perform searching for clauses by holding indexes on the clause name and also the first argument in the clause, thus improving the search performance of sequential systems considerably [Warren 88b].

2.3.4.3. OR Parallelism

OR parallelism describes the simultaneous execution of alternative versions of the evolving query as a separate process. This may involve alternative versions of rules at a high level in the solution tree or of base predicates at the leaves of the tree. The concept of OR parallelism replaces backtracking in a sequential system. Instead of following one branch of the solution tree, having marked the backtrack point, the system spawns separate processes for each alternative, and these processes become candidates for simultaneous execution. The query

<- a(x)

to the following rule base

a(x) <- b(x) or c(x)

would produce the OR tree as shown in Fig.2.8.

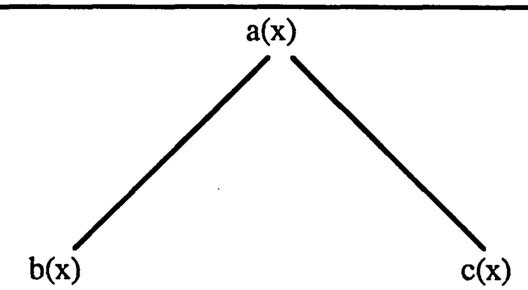


Fig. 2.8 - OR Tree

A further example of an OR tree that includes conjoined expressions is given in Fig.2.9. This represents the query

<- a(x y z)

which has been put to the rule base:

a(x y z) <- b(x) and c(y z)

b(x) <- d(x) or e(x)

c(y z) <- f(y) and g(z)

The two alternative subexpressions contained in the definition of b(x) lead to the branching of the OR tree.

Two points emerge from this high level description of OR parallelism: first it can be seen that a move to this approach makes the language model set based. Unlike Prolog any OR parallel system involves a search for the full set of bindings which satisfy the query. The user may wish to curtail the search after a given number of bindings have been produced but the concept of handling binding sets remains. This is not generally true of sequential systems: in Prolog the search suspends once the first binding is produced and in order to obtain the full set the system has to be repeatedly prompted or the instruction has to given within the program by the use of a construct such as "findall" or "bagof" [Bratko 86].

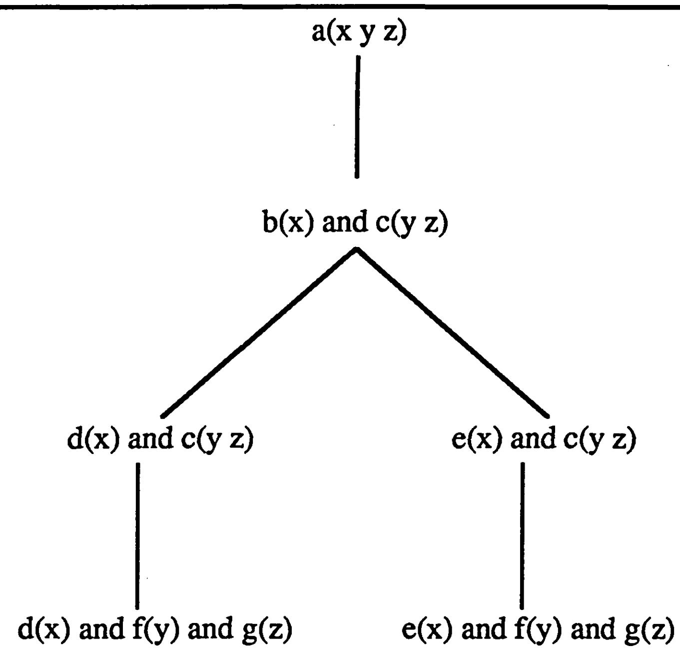


Fig. 2.9 - OR Tree

The second aspect is that the separate processes created during OR parallel execution are conceptually independent of each other, as bindings represent proper alternative values; this can be seen in the OR tree representation in Fig.2.9 where the two OR processes result in the alternative binding sets for x,y,z. This means that each process can run to termination without reference to any other process and no forms of synchronisation or suspension are needed. However in the same way that backtracking involves returning to a previous computational state, each OR process has to inherit the environment of its parent, and variables that were bound at the time of OR process creation must remain bound in the new offspring processes. This creates an implementational problem in that although each process is independent, it shares a common parental environment. If this is held in a shared memory provision must be made to ensure that new bindings made by child processes are private to that child (and any subsequent offspring); on the other hand if the parental environment is to be copied for each child process, considerable overheads may be introduced. (There is a third possibility, ie that a process recomputes its parental environment when it is set up: this will be discussed in Chapter 3.1.3.3. in relation to the Delphi project [Alswawi 88]).

For many OR parallel systems one of the main implementational concerns is not with defining enough parallelism to execute but containing it. In programs where there are a large number of alternative rule and base predicate definitions, the number of OR processes produced in the course of query response may swamp the computational resources available. This will be looked at in more detail in the context of architectural requirements. Thus the issues involved in the definition of an OR parallel system are implementational rather than theoretical because of the independent nature of the processes. Representation of the binding environment in both shared and non shared memory models is of importance, and load balancing, ie control of distribution of OR processes, is necessary in either system.

2.3.4.4. AND Parallelism

AND parallelism occurs when the conjoined subexpressions in the body of a rule are executed in parallel. To return to the rule base from the previous section, ie
a(x y z) <- b(x) and c(y z)
b(x) <- d(x) or e(x)
c(y z) <- f(y) and g(z)
when the query
<- a(x y z)
is entered into the system, b(x) and c(y z) can be executed in parallel and similarly f(y) and g(z) are candidates for parallel evaluation. This is known as AND parallel execution. When alternatives are also present in the rule base an AND-OR execution tree can be defined for the query evaluation. Both conjoined subgoals and alternative calls are expressed as separate nodes in this tree as shown in Fig.2.10.

The line linking two arcs indicates that the offspring nodes are conjoined expressions. Although d(x) and e(x) have been shown to be independent computations because they represent alternatives, this is not true of b(x), c(y z), f(y) and g(z). Each of these provides part of the response to the query

<- a(x y), and their results must be communicated to the parent process in order to combine them. This is the crucial difference between AND and OR parallelism: in the latter the communication in the solution tree is one way, ie downwards; in the case of AND parallelism it has to be bidirectional, ie between parent and child and vice versa. Thus implementations of AND parallelism must incorporate controls for synchronisation and combination of results. The AND-OR tree for the small clause set shows four leaf nodes which could be candidates for parallel execution under a system implementing AND and OR parallelism. The OR tree that is produced by the same query has already been discussed in the context of OR parallelism (see Fig.2.9).

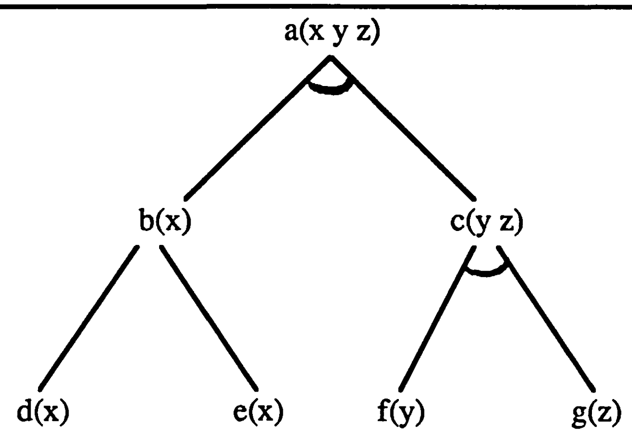


Fig. 2.10 - AND-OR Tree

In the example given above the two conjoined subexpressions contained mutually exclusive variables. However the situation commonly occurs where variables are shared between two or more subexpressions. The shared variable is an important feature in most logic languages programs and in the case of recursive definitions it is necessary for passing binding values to the next level in the solution tree. In the following example
a(x y) <- b(x y) and c(x) and d(y)
it is not sufficient for b(x y), c(x) and d(y) to return values for x and y to the parent process, they have to be checked for consistency. In a program where the predicate names b, c and d represent sets of alternative base predicates it becomes apparent that this consistency checking requirement is directly comparable to the Join operation in relational databases. Fig.2.11 shows the AND-OR tree for this query including the sets of returning bindings.

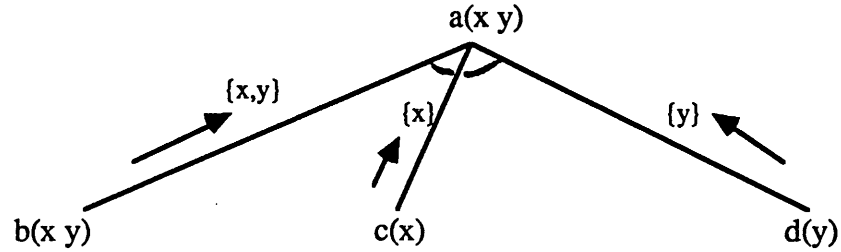


Fig. 2.11 - AND-OR Tree with Shared Variables

To consider a further example the clause set introduced in Chapter 2.2.1 and repeated in Chapter 2.2.3 shows the logic program and the directly comparable relational database. The Join operation illustrated in Fig.2.3:
JOIN (Relation1, Relation2)
is analogous with the query
<- teaches(x, y)
and the resulting AND-OR tree shown in Fig.2.12 shows that the values returned for "course" from the leaf nodes have to be checked for consistency and only when this is established can a valid binding set of {x,y} be constructed.

If full AND parallelism is incorporated into the parallel logic system the full consistency checking operation has to be performed on bindings produced for shared variables. The management of the shared variable problem is the main implementational issue in AND parallelism and causes considerable difficulties.

There are two aspects to shared variable management; the first is recognition of shared variables and secondly containment of the consistency checking phase. Automatic recognition of shared variables is not straightforward as the differing pattern of variable instantiation affects this. Returning to the simple example given above, as can be seen in Fig.2.11 both the variables x and y were shared between two subexpressions; however if x is instantiated at query insertion time, there is no need to include it in a consistency checking operation, y will be the only variable involved. In this way identification of shared variables at run time will take into account the particular pattern of instantiations throughout query evaluation.

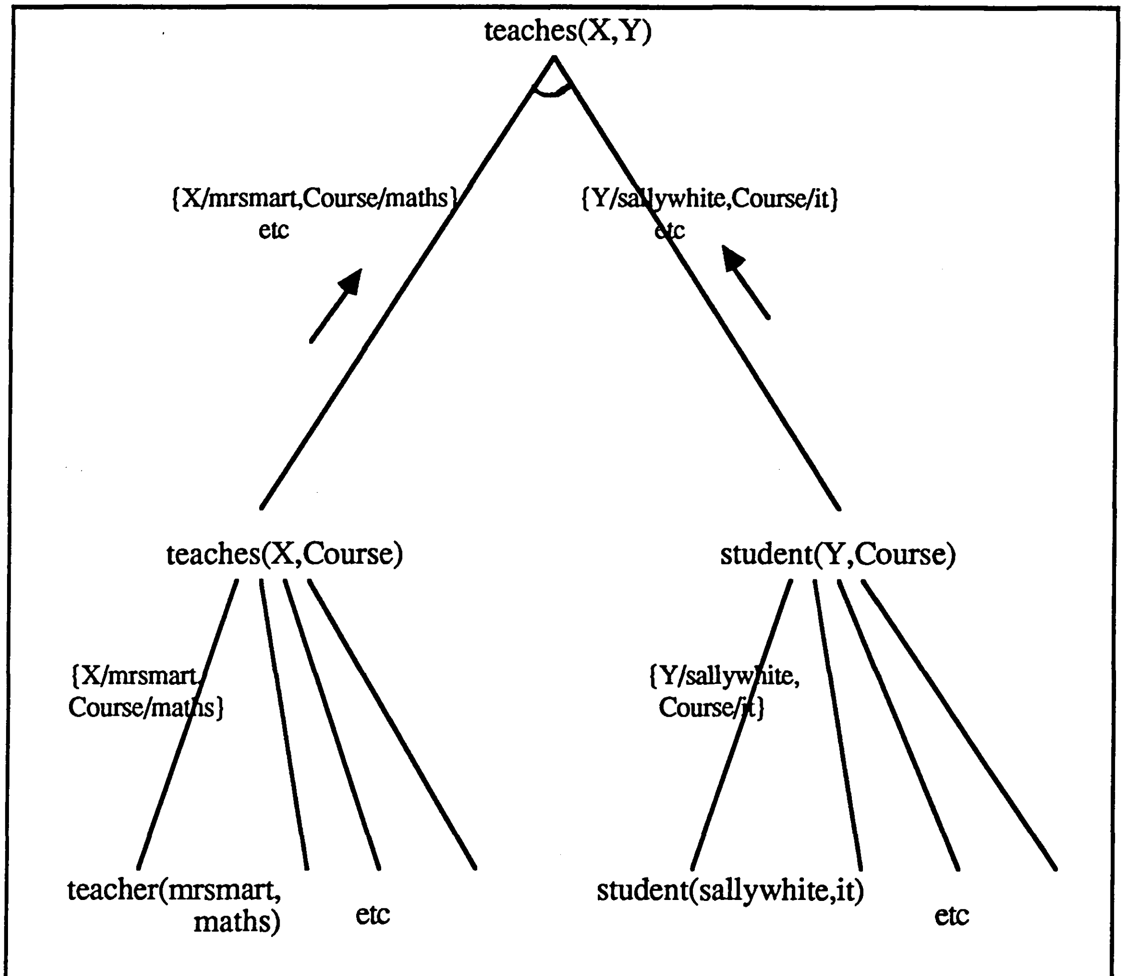


Fig. 2.12 - AND-OR Tree for Student-Teacher Program

Alternatively shared variable recognition can be done at compile time. The run time flagging of shared variables involves complex algorithms and thus imposes computational overheads on the system; the compile time approach does not create extra processing at run time but neither does it lead to such a finely tuned system.

The reason for identifying the shared variable is to implement a form of producer-consumer parallelism which avoids the need for a full scale Join operation. Most projects concerned with the AND parallel execution have proposed some form of this type of pipelined producer-consumer parallelism. Of course with this approach if there is only one consumer to a producer no AND parallel execution is possible; only where one producer is supplying values to several consumers can benefits from AND parallelism be gained. In the above example b(x y) would be designated as producer and would execute first passing values for x and y to c(x) and d(y) which could then run as separate AND processes. Examples of systems using automatic detection of shared variables will be discussed in Chapter 3.1.2.2.

Automatic detection of shared variable dependencies creates theoretical problems and imposes computational overheads: a more pragmatic approach is to include "mode" declarations which are installed by the programmer to indicate whether a variable is to be a producer (input mode) or a consumer (output mode). These mode declarations are used to control the parallel execution of subexpressions, and have been employed in a number of systems. Other program annotations may be available to indicate when the system is to operate in parallel mode for both AND and OR processes. Whilst recognising that the ultimate aim is the automatic production of safe and efficient parallel logic language execution many researchers have felt that the use of programmer control over parallel execution provides a temporary means of exploring many of the complex issues involved. Examples of systems using this approach can be seen in Chapter 3.1.2.3. As will be shown this project has taken the other stance, ie to maintain the transparency of parallel execution even if the resulting system is less precise in its implementation.

2.3.4.5. Stream Parallelism

A specialised form of shared variable producer-consumer parallelism is known as "stream" parallelism. This refers to the method of passing large structures (typically lists) from one AND process to another as they are formed - the first process adding values to the end of the list at the same time as the second process consumes them by removing them from the front. This is a typical use of pipelining in processing and involves the usual controls on synchronisation. It is frequently used in the concurrent committed choice languages [Shapiro 83], [Clark 83], [Ito 83], [Ueda 86].

2.3.5. Potential Performance Benefits from Parallel Execution

It has been shown that the logic language paradigm includes two abstract level concepts, ie AND and OR parallelism, which would allow a parallel computation model to be designed. Before looking at examples of systems which have incorporated the concepts of AND or OR parallelism it is worth considering the theoretical advantages that the inclusion of parallel execution can bring, ie what are the potential performance advantages to be had in implementing a parallel model. In general the amount of exploitable parallelism is highly application specific. In the type of programs written to implement the deductive database type of system, the scope for OR parallelism is considerable as these systems usually contain many alternatives of both rules and base predicates. An example of this type of system is the Molecular Protein Database project of the Imperial Cancer Relief Fund. This is written in Prolog and at present holds information on almost 300 proteins in the form of approximately 300 rules and 5,000,000 base predicates [Rawlings 87], [Rawlings 90]. The type of program which is largely deterministic in nature is likely to offer much more limited scope for OR parallel execution. Examples of the latter type of program include the traditional append program for concatenating two lists (see Appendix C).

Program	Mean Degree of Parallelism	Max. Degree of Parallelism
append	2	2
member	2	2
atlas	811	2548
mutation	78	255
map0	425	920
map1	106	249
map2	37	86

Fig. 2.13 - Measurements of Potential OR Parallelism

The work by Ciepielewski in measuring the amount of potential OR parallelism in a set of small benchmarks gives some idea of the range of potential speedups to be obtained [Ciepielewski 86]. His results dealt at an abstract level taking no account of communication overheads and assuming that the system had unlimited processing resources; thus they represent the theoretical maxima for performance benefits under idealised conditions. Examples of the results obtained are given in Fig.2.13. The maximum speed up that could be obtained for a query response is directly related to the mean degree of parallelism; the maximum degree of parallelism is determined by the maximum number of processes concurrently active during an execution run, ie it represents the number of processing elements required to produce the ideal performance benefit.

These results proved helpful in the decision to concentrate on this form of parallel execution in this project as they showed that realistic speedups could be achieved for the type of applications under consideration. The database type of programs used in these tests were felt to provide appropriate models for the larger and more complex knowledge based systems likely to result from the field of applied artificial intelligence.

It would be helpful to see the same form of theoretical analysis for generalised AND parallelism but as the implementation of this form of execution is dependent on the details of the system it is not possible to obtain such a clear picture. However the analysis of Pure Logic Language programs (see Appendix D) has led to the conclusion that the amount of potential AND parallel execution to be obtained may be fairly restricted for a large number of programs. This tentative conclusion is reinforced by the results given for the PEPSys system where only small performance gains were obtained from some of the benchmark programs: although it is not directly stated it is likely that the poor potential for parallel execution is due to the lack of OR parallelism within the programs [Chassin de Kergommeaux 89]. In general the typical number of conjoined expressions in a rule is in single figures; in a program where each rule body contains two conjoined expressions, and each rule has two alternative definitions the number of leaf nodes in the search tree is $4^{(n-1)}$. For a deep tree this will lead to a large number of processes where AND-OR parallelism is employed: a tree with five levels will produce 256 leaf nodes to be evaluated, whereas for a tree of depth ten the number rises to more than 250,000. However it is necessary to add the serialising effect of the shared variable handling mechanism, and this has been shown to limit severely the amount of actual AND parallel execution that can take place [Kale 88b].

2.4. Summary

The chapter has taken an overview of the type of system where the inclusion of an inferencing mechanism is likely to involve some form of logic interpreter. The methods by which this can be integrated into a data handling system have been looked at, and the implementation of the logic programming language Prolog described. The declarative nature of logic languages appears to offer scope for the development of parallel execution models and the mapping of these models onto multiprocessor architectures. The sources of parallelism in logic languages have been discussed and the problems in the implementation of parallelism identified. The next chapter will look at some examples of parallel logic systems and architectural proposals for parallel hardware.

Chapter Three

Parallel Logic Systems and Associated Architectures

3. 1. Parallel Logic Language Systems

3.1.1. Introduction

As the discussion in Chapters 1 and 2 has indicated, this project has been based on the approach that the implementation of an OR parallel model for the Pure Logic Language is appropriate for the applications area under consideration. The inclusion of AND parallelism has been discounted at present because it is not clear that it would provide major performance benefits. (Examples of this type of system are described in Chapter 3.1.2.2). It is therefore appropriate to concentrate attention on the work that been done in the implementation of OR parallel systems by the major research groups. Brief reference will be made to those proposals concerning the efficient implementation of AND parallel models.

There is one group of languages in which the OR parallel approach plays a very limited part: this is the set known as the committed choice languages which operate a concurrent process model. They include Concurrent Prolog, PARLOG and Guarded Horn Clauses [Shapiro 83], [Shapiro 86], [Clark 83], [Clark 86], [Ueda 86]. These languages differ in concept from the "parallel" logic languages, and they address a different applications area, typically being used to implement operating systems, process control applications and other programs where the communicating sequential process paradigm is appropriate. There is an increasing amount of work being undertaken on their implementation and they represent one of the important developments in logic programming. However this is outside the scope of this project as they do not present a suitable model for the implementation of OR parallelism.

This section will look at several examples of parallel language systems from the language implementation viewpoint. A number of languages have been employed on specific multiprocessor architectures, and this aspect will be discussed in the next major section which deals with architectural issues. The examples of systems described below is not intended to be an exhaustive list but represents some of the major work

System	Language	AND Parallelism	OR Parallelism	Control Annotations	References
BC Machine	Prolog	No	Yes	No	[Ali 88a], [Ali 88b]
Delphi	"Pure" Prolog	No	Yes	No	[Alshawi 88]
Gigalips/ Aurora/ ANLWAM	Prolog	No	Yes	No	[Butler 88], [Calderwood 88], [Hausman 89], [Lusk 88], [Warren 88a]
PEPSys	Prolog (modified)	Independent	Yes	Yes	[Ratcliffe 87], [ChKergommeaux 88], [ChKergommeaux 89]
BRAVE	"Brave"	Full	Yes	Yes	[Reynolds 87a], [Reynolds 87b]
Gigalips/ Extended ANLWAM	Prolog	Full	Yes	Specialised Predicate Calls	[Carlsson 88]
AND	"logic"	Restricted	Yes	Compile Time Analysis	[Chang 85]
AND/OR	"logic"	Restricted	Yes	Run Time Analysis	[Conery 83], [Conery 85], [Conery 87]
RAP	"logic"	Restricted	No	Compile and Run Time Analysis	[DeGroot 84]
PSOF	"logic"	Restricted	Yes	Compile and Limited Run Time Analysis	[Hwang 89]

Fig. 3.1 - Summary of Parallel Logic Systems

that has been reported in the field. The table (Fig.3.1) summarises the salient features in the systems under consideration. References to other research work in the field of parallel logic systems include [Beer 86], [Biswas 89], [Bosco 89], [Cheese 87a], [Cheese 87b], [Diel 86], [Kale 88b], [Naish 88], [Odijk 86], [Rhaman 88], [Shaw 85], [Stolfo 87], [Wise 86].

It can be seen from the table that the Gigalips system is responsible for more than one system. This joint project involves research teams from the United States (Argonne), Britain (Bristol) and Sweden (SICS, Stockholm), and results from an informal amalgamation of separate work already underway in the different institutes. Some of the work has incorporated a form of AND parallelism but the main emphasis has been on OR parallel logic languages, resulting in AURORA and ANLWAM versions of Prolog. The Gigalips projects are also involved in the design of multiprocessor architectures, and proposals for shared memory and "distributed" memory machines will be looked at in Chapter 3.2.4.

Before looking at the implementational issues involved in OR parallel execution attention is focussed on the systems which also incorporate some degree of AND parallelism. The question of memory management in these systems is complex as not only do considerations of shared versus non shared data structures arise (as they do in OR parallel systems) but process synchronisation has to be included. This aspect is only discussed in outline. The memory management of OR parallelism within the AND-OR systems is looked at in Chapter 3.1.3 in the context of pure OR parallel systems.

3.1.2. AND Parallel Systems

3.1.2.1. Introduction

The AND parallel systems can be subdivided into those which implement AND parallelism in an automatic fashion, ie it is transparent to the programmer, and those systems where the programmer has to specify when parallel execution is to take place. As will be shown in the next section (Chapter 3.1.2.2) and the analysis contained in Appendix D for the Pure Logic Language program, automatic generation of "safe" AND parallelism is not necessarily going to offer real performance benefits. The use of programmer introduced annotations to specify parallel execution of conjoined subgoals is looked at in the following section. In these systems it is possible to incorporate knowledge about the program's execution behaviour and architectural considerations.

The terms "independent" and "restricted" are commonly used to describe the type of AND parallelism employed: these are used in the same sense to indicate that parallel execution of conjoined subgoals is only permitted when there are no shared uninstantiated variables involved.

3.1.2.2. Transparent AND Parallelism

The four systems listed at the bottom of the table (Fig.3.1) represent examples of work done in incorporating automatic detection of "safe" AND parallelism into logic language systems [Conery 83], [Conery 85], [Conery 87], [Chang 85], [DeGroot 84], [Hwang 89]. The work is relevant to this project only in as much as it indicates the type of data dependency analysis that would need to be included if the parallel Pure Logic Language system were to move towards an AND-OR model and retain its declarative nature.

The original proposal by Conery used an AND-OR process model [Conery 83], [Conery 85]. It defined a parallel execution scheme based on message passing between processes which were generated automatically by the system. Because AND processes were defined, messages to implement synchronisation and suspension of processes as well as transmission of bound variable values were required. The system automatically detected shared variables and used an ordering algorithm to work out the pattern of subgoal evaluation. This variable dependency checking operation was performed at run time whenever a new conjoined goal list was produced.

The original ordering algorithm proved to be unable to cope with certain cyclic rule definitions and subsequent work by Conery refined and extended it [Conery 87]. However the overheads involved in running a full dependency analysis check throughout query evaluation led to proposals to perform the operation at compile time.

Work on compile time analysis by Chang, Despain and De Groot has lead to the concept of storing data dependency graphs which indicate the patterns of data flow between subgoals [Chang 85]. The first proposal used these graphs which were set up at compile time to work out shared variable dependencies during program execution. The problem with this approach is that in order to ensure "safe" AND parallel behaviour, a conservative view on parallel execution has to be taken. The following example demonstrates this. The rule which defines one aspect of Bill's social behaviour is:

has_date("bill", x) <- likes("bill",x) and available(x,day) and enjoys(x,activity) and open(activity, day)

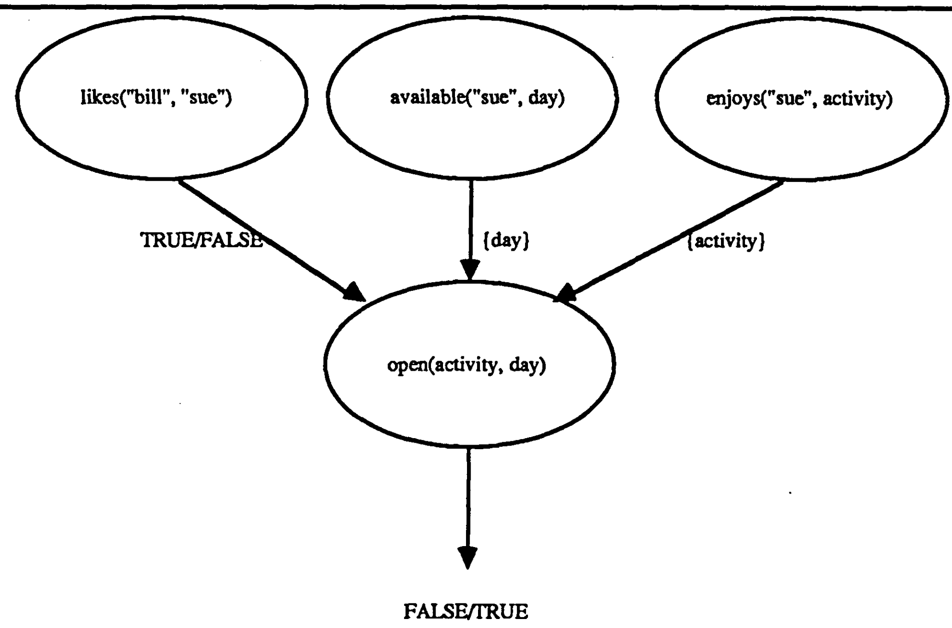


Fig. 3.2 - First Dependency Graph

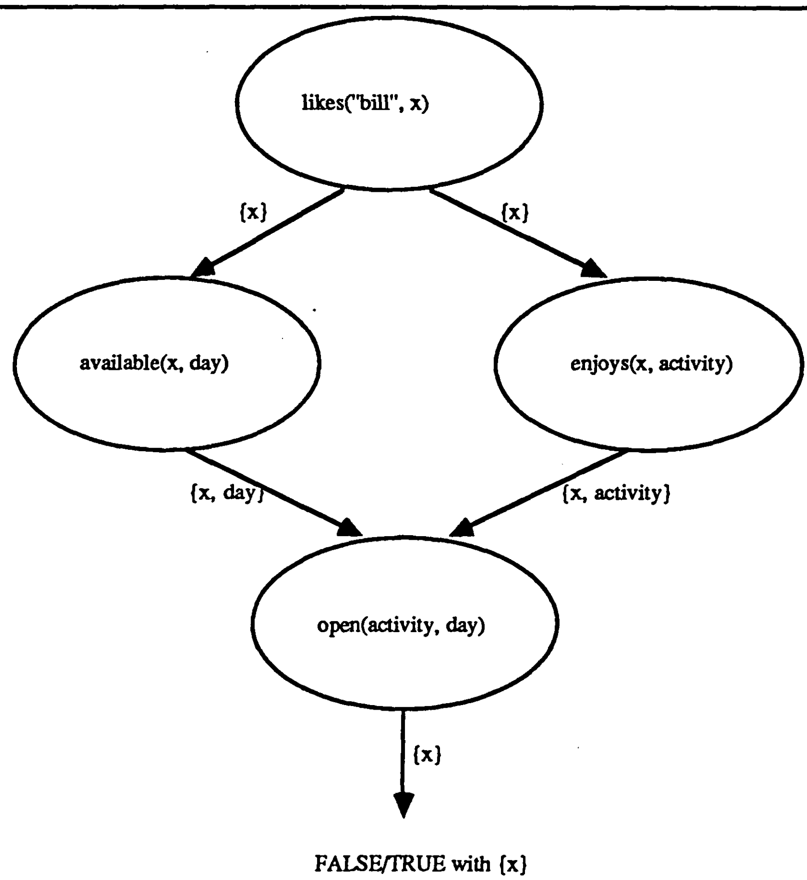


Fig. 3.3 - Second Dependency Graph

If x were instantiated at the time of query insertion, the data dependency graph would indicate that the first three subgoals could be solved in parallel (Fig.3.2). However if x is not instantiated it would appear

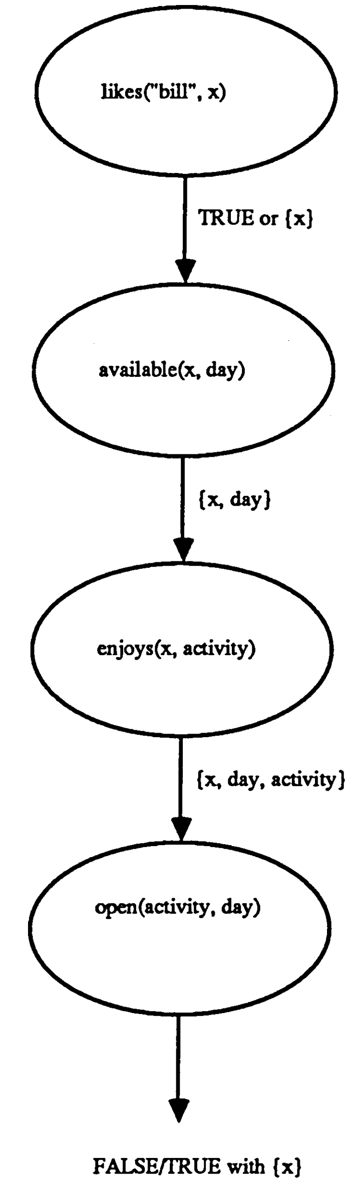


Fig. 3.4 - Third Dependency Graph

that the graph should be reduced to Fig.3.3. Unfortunately even this limited amount of AND parallelism is not safe, as it is not necessarily true that the call to likes("bill" x) will bind x - it is possible that the database contains the base predicate likes("bill" x), ie Bill likes everyone! Thus for this rule the safe data dependency graph produced at compile time has to define a serial implementation of the rule (Fig.3.4).

These two systems demonstrate the dilemma involved in the automatic detection of AND parallelism: accuracy can be sacrificed for run time efficiency, or computational overheads can be introduced in the hopes that the added parallelism will be worthwhile. The problem is still under active consideration and several proposals have been put forward for mixed schemes which incorporate both compile and run time checking: the two final entries in Fig.3.1 represent examples of this method [DeGroot 84], [Hwang 89]. The overheads in this approach are considerably less than those in the original compile time checking schemes.

3.1.2.3. Programmer Control of AND Parallelism

The three systems listed above which involve programmer control over AND parallelism are the ECRC PEPSys system, the BRAVE language developed at Essex, the extended ANLWAM scheme from the Gigalips project [Ratcliffe 87], [Chassin de Kergommeaux 88], [Chassin de Kergommeaux 89], [Reynolds 87a], [Reynolds 87b], [Carlsson 88]. The first two proposals are looked at in more detail as they represent examples of two differing approaches to controlled AND parallelism. In PEPSys the programmer must prevent subgoals simultaneously competing to instantiate shared variables, whereas in BRAVE this is allowed and the system provides for the resultant consistency checking operation. The extended ANLWAM system incorporates a stream type of parallelism by the inclusion of specialised predicates rather than program annotations [Carlsson 88].

The PEPSys project running at ECRC in Munich is a major research effort involving multiprocessor architectures for parallel logic programming as well as theoretical work on the language definition issue. The aim is to produce a multiprocessor system which will give worthwhile performance benefits with large Prolog programs [Ratcliffe 87], [Chassin de Kergommeaux 88], [Chassin de Kergommeaux 89].

The parallelism is controlled by dividing the program into modules, serial and parallel. Serial modules represent standard Prolog code, including extra logical features if required, and are executed in a sequential manner using the normal Prolog backtracking mechanism. Parallel modules contain self referencing code, ie predicates defined within these modules can only call predicates belonging to parallel modules. No side effects are permitted within this code. The interface between serial and parallel modules is implemented by the built-in predicates "setof", "bagof", "oneof", the latter providing the semantics for the return of one result only which will be the first in time to be produced.

Within parallel modules AND parallel execution is labelled by the programmer where the programmer is sure that independence of variables is guaranteed, eg

a(X,Y) :- b(X) # c(Y).

indicates that the b(X) and c(Y) are candidates for parallel execution. OR parallel execution of alternative versions of predicates is assumed for those contained in parallel modules. The intention is that knowledge of the multiprocessor architecture plus the type of computations involved in a given program is to be used by the programmer to produce the most efficient code for a particular application.

The PEPSys system is based on a modified Warren Abstract Machine, known as the PAM. This means that not only is the code compiled into groups of WAM instructions but a PAM contains the memory management structures as defined for the WAM (see Chapter 2.2.6). Each processor or virtual processor has its own PAM so that on each processor a process operates independently in its own environment. However data (bindings or goal lists) may be inherited from other processes and thus may appear in the PAM of other processes. This situation is handled by marking data as either belonging locally or by indicating which non local process contains the necessary information. When some non local data is required, access to the PAM of the other process or processes must be allowed. In a shared memory implementation this is straightforward, but in a distributed memory system this access will involve copying data from one processing element to another. Thus although conceptually the model does not require a shared memory implementation, because of the references within a process' PAM to data from a number of other processes, a shared memory implementation is likely to be more attractive.

The PEPSys system has been tested in three stages:

a) the implementation of the abstract model showing the parallel behaviour of different programs, but ignoring such issues as communications overheads and limitations in computational resources,

b) a simulation of the system running on a new design for a "multi-cluster" machine,
c) a multiprocessor implementation using a Siemens MX500 multiprocessor. The Siemens MX500 machine is similar to the Sequent Balance 8000, and both comprise eight processors and a shared memory (16 Mbytes in this instance) using a common bus. The shared memory is logically divided into partitions for each processor: each partition holding the stacks relating to the PAM for the individual processor. Communication is achieved by allowing a processor read-write access to its own stacks but read-only access to those of others.

The simulation results given for a number of benchmark programs indicate that the performance predicted for the abstract analysis should be possible in a "real" machine. However the amount of performance benefit is highly application dependent: much of the analysis for the reasons for this is concerned with the "size" of each process, ie the granularity of the system, as a large number of short lived processes appear to degrade the performance. This would indicate that the overheads of process creation are sizeable in relation to processes only performing a small number of inferences. Unfortunately no analysis is given of the separate roles played by AND and OR parallelism in the benchmark programs, nor is there any comment on the effectiveness or otherwise of the programmer's use of parallel control structures, other than the general conclusion that the setting up of short lived processes should be avoided, presumably by restricting parallel definitions in some way. However it would appear that in general the OR parallel execution is providing the basis for most of the performance benefit [Chassin de Kergommeaux 89].

The BRAVE language system is the result of research carried out at the University of Essex [Reynolds 87a], [Reynolds 87b]. Unlike the approach taken in the PEPSys the program is not divided into parallel and sequential modules, but within each rule definition explicit control of parallelism must be specified; conjoined expressions are candidates for parallel execution if the following syntax is used:
a(X,Z) :- b(X,Y) & c(Y,Z).
Serial execution is indicated as:
a(X,Z) :- b(X,Y), c(Y,Z).

OR parallel execution is defined in the alternative versions of rules by the ":" notation, whereas the terminator "." indicates sequential execution in textual order as with Prolog. Thus

a(X,Y) :- b(X,Y).
b(X,Y) :- b1(X,Y):
b(X,Y) :- b2(X,Y):

indicates that in order to satisfy the goal a(X,Y)., the alternative definitions for b(X,Y) can be evaluated simultaneously. Had the terminator been the standard Prolog ".", normal sequential evaluation would take place, ie b1(X,Y) would be tried before b2(X,Y).

The example given to indicate AND parallel execution shows that the programmer is not forced to specify serial execution when shared variables are encountered. Reynolds discusses proposals for incorporating a consistency checking mechanism that operates on the different binding values as they are returned from conjoined subgoals, rather than holding onto all the sets in memory and performing a full Join operation [Reynolds 87a]. However it may in fact be advisable to direct this process by use of the serial conjunction annotation in the case that the first subexpression is likely to bind variables and thus reduce the search space for subsequent subgoals. There is clearly considerable scope for developing programmer heuristics for the best method to control execution through the use of AND and OR parallel/serial annotations in this and other similar proposals.

BRAVE is implemented using a compiled system based on the Warren Abstract Machine known as the BAM. The BAM holds memory management structures and compiled code in the same way as the WAM (see Chapter 2.2.6) and each process has access to this global abstract machine thus providing the basis for a shared memory implementation of the system. The BAM code instructions include a "compose" operation to perform the consistency check in the case of full AND parallel execution of conjoined subexpressions which share variables. By starting the consistency checking task as soon as each subgoal has returned its first set of bindings the necessity to hold the entire variable binding sets in memory at one time is reduced. Reynolds discusses the situations in which it is appropriate to call for full AND parallelism (including a consistency checking operation) as opposed to the serial implementation which can involve co-routining of subgoals [Reynolds 87a]. The programmer is responsible for determining which form of parallel execution is likely to lead to the best performance.

The system has been implemented on two shared memory machines, and results are given from a prototype machine with three processors and a common memory, and the GRIP machine [Reynolds 87b]. The system has also been implemented on a Transputer grid in which the global memory has to be spread throughout the local memories of the Transputers. Access to non locally held data involved message passing and not surprisingly communications overheads proved high for this method of implementation [Reynolds 87b].

3.1.3. OR Parallel Systems

3.1.3.1. Introduction

It can be seen from the table in the Chapter 3.1.1 that the inclusion of OR parallelism is implemented in all but one of the systems under consideration. This involves the concept of treating alternative branches of the search tree as separate processes and executing them simultaneously. Fig.2.9 in Chapter 2.3.4.3 showed a simple OR tree for an expression involving conjoined expressions. The tree given in Appendix E as an example of the search space produced by a test Pure Logic Language query to the reduced benchmark database shows an OR tree which contains approximately forty OR processes. When the same query was used with the proper benchmark database more than seven hundred OR processes were generated. It is generally accepted that this provides a solid basis for performance benefits in a wide range of applications [Kale 88a]. The theoretical issues of variable dependencies do not arise in pure OR parallel systems. However the representation of alternative bindings for the same variable has to be tackled, as do mechanisms for the control of OR parallel execution. This latter task can be performed either by restricting the number of OR processes formed or by efficient scheduling methods.

When a child OR process is created it has to inherit the "binding list" and "environment" from its parent. These are directly comparable to the situation that exists in a sequential implementation, OR processes being created as an alternative to the storage of backtracking points. The binding list refers to the set of bound variables which are present in the query or have been introduced by subsequent evaluation of subgoals. The environment is the goal list that exists at that particular point in time. This may be implemented in various ways but is likely to involve some form of linked list of unpredictable size. In Chapter 2.2.6 it has been shown how the linked chain of environment frames is used to represent the state of the goal list in the WAM. The passing of this environmental information from parent to child processes can be achieved in three ways:

a) sharing access to the environment,
b) copying the environment from parent to offspring,
c) recomputation of the environment by the offspring.

Of these alternatives a) assumes that some form of global or shared memory is available. Most OR parallel proposals have developed schemes for sharing access to environments because it has been felt that the cost of copying the required information is too great [Kale 88a]. Although most proposals discount the full copying operation at process creation time, any actual implementation that uses a non shared memory machine, eg Transputer based testbed for BRAVE (see Chapter 3.1.2.3), must inevitably perform some copying as execution proceeds and access is required to data which is held on remote processing elements. The implementation of a full copying of environment prior to process creation is implemented in the SICS BC-Machine proposal [Ali 88b]. One scheme has been found that relies on recomputation as a means of giving child processes access to parental environments: this is the Delphi proposal [Alshawi 88]. As will be seen in Chapters 5 and 6 the proposals for the parallel Pure Logic Language are based on the view that a form of mixed copying and recomputing the process environment at the time of process creation does not necessarily produce unacceptable overheads if the physical architecture can be designed to optimise this method. This approach is conceptually close to that taken in the SICS BC-Machine system and the Delphi project but the actual method of implementation is different.

The following sections look in more detail at various OR parallel systems. The first group rely on a process having a mixed form of access to data, some shared with other processes and some local. The second group consists of the two schemes referred to above which are specifically intended to operate in a non shared memory context.

3.1.3.2. Data Sharing OR Parallel Systems

When environmental data, ie the binding environment and the goal list, is made common to two or more processes running on different processing elements by the use of a shared memory, two aspects require attention. The first is contention for access to the memory: this depends on the type of locality of reference shown in the program and on the hardware design (see Chapter 3.2). The second aspect is the representation of alternative bindings for the same variable.

In the rule base containing the following definition
a(x y) <- b(x y) or c(y) or d(y z)
if the query
<-a(x y)
is put with x instantiated to a value, the three OR processes have the following bindings requirements:
b(x y) inherits the binding for x and will attempt to bind y,
c(y) has no interest in x and will attempt to bind y,
d(y z) has no interest in x and will attempt to bind y, and also a locally introduced variable z.

If any of these three processes spawn further child processes these offspring will need to operate in the environment of their parent but may also require access to the grandparent's binding environment in the case of b(x y)'s descendents. Bindings made by any of the three offspring processes for y are independent of each other and have to be held separately. When all processes are operating on separate processing elements but sharing a common memory, means of representing this hierarchy of binding environments must be found, as the copying of parental binding lists for each new process defeats the object of maintaining a shared memory. If each process is given a designated list or "window" for the bindings performed locally and access to the location of its parent's window, each time it needs to bind a variable it has to check up the tree of windows to ensure that the variable has not been bound by an ancestor. Fig.3.5 shows a representation of the OR tree and the binding window which is associated with each process. If during unification descendants of Process 2 need to check whether x and y are bound, they must search both the binding list for Process 2 and for its parent, Process 1. It is more efficient for an OR process to be passed a pointer to the start of the appropriate chain of binding windows than it is to copy the total bound variable list for each process. Of course where no shared memory exists the copying of necessary variable bindings has to be performed.

Many OR parallel implementations have used variations on this theme. Because there may be situations where large number of variables

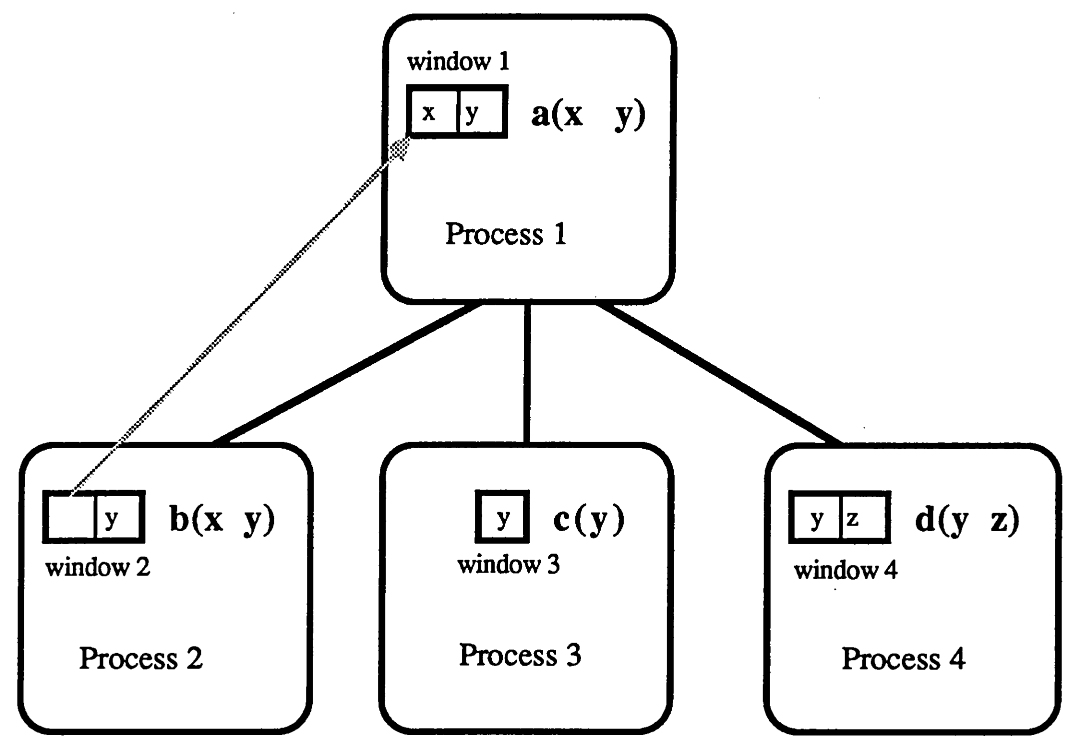


Fig. 3.5 - Representation of Binding Windows

have to be included in a binding window (eg when a lengthy recursive call involves many introduced variables) some implementations have employed indexing and hashing methods to speed searching for bindings in these lists. These include the hash windows of PEPSys [Ratcliffe 87]. The details of these proposals are not of prime importance to this project and doubt has been cast on the advantage to be obtained by using them [Kale 88a]. Other proposals define a binding list for each processor rather than each process [Warren 84]. In systems produced by the Gigalips projects, eg Aurora and ANLWAM Prolog, the solution space has been divided into "public" and "private" sectors [Butler 88], [Lusk 88]. Within a private sector (lower down the solution tree) a standard Prolog sequential systems works and uses normal backtracking methods operating on a standard binding representation; the public area defines work that is to be shared out and thus requires management of shared binding environments. The optimal division of public and private sectors is employed as a process scheduling method and is a matter for run time adjustment [Butler 88], [Calderwood 88]. Further refinements of this approach divide the search into three sectors, public, private and an intermediate level known as "favoured" [Kale 88a].

3.1.3.3. Non Shared Data OR Parallel Systems

Finally the systems which have been proposed for non shared memory architectures are looked at. These relate closely to the approach taken in this project in which it has been assumed that because of scalability problems, shared memory multiprocessors with large numbers of processing elements are not viable for this type of application. If all memory is to be local to the individual processing elements, processes executing on different processing elements must "share" data by copying it, or alternatively the junior process must recompute some or all of the data. An example of this type of scheme is discussed later in this section.

The implementation of OR parallel Prolog on the BC-Machine at the Swedish Institute of Computer Science uses a copying of data method [Ali 88b]. Standard Prolog is used as the language model and is implemented in a sequential fashion using compiled WAM techniques. When alternatives are encountered, the offspring OR processes are allocated to remote processing elements where the standard Prolog system performs the computation. The reason that this method can be used efficiently lies in the communication between separate processing elements and the load balancing of work throughout the machine. Communication is achieved by dividing the processing elements into groups of "masters" and "slaves": as execution of a process takes place in the master the environment is continually written into the memories of each of the slaves. Thus when OR processes are spawned by the master there are a number of other (slave) processing elements which contain the same environmental information. Evaluation of child processes can proceed in these slave processing elements almost immediately as there is no large delay due to copying overheads; the only information which needs to be accessed at this stage is the data on which branch, ie OR process, the slave node is to handle. This is achieved by using a small control memory into which each master writes "control frames" for each OR process which needs to be allocated to a slave.

The BC network system is regarded as a broadcast net as communications are implemented on a one to many basis, ie one master to many slaves. The actual proposals for hardware implementation will be looked at in section on multiprocessor architectures (Chapter 3.2.4.4).

The pattern of masters and slaves is a hierarchical one altering in time during query evaluation. Nodes which start out as slaves to the first master processor may become designated masters and achieve their own group of slaves as the amount of work, ie separate OR processes, increases.

The efficient implementation of the software execution clearly depends on the method of scheduling work to the processing nodes, and obtaining the correct balance between masters and slave numbers. In the actual implementation work is not disseminated at the first possible opportunity. The load balancing parameters are k (the threshold of the number of locally created OR processes), g (the number of processor groups created on each reconfiguration) and m (the threshold of the number of idle processing elements). When processing starts the first master executes a standard sequential Prolog program storing up choice points, ie OR processes, until k choice points are reached. At this stage the k control frames are written to the global memory and the system is partitioned into g groups with a master in each. Because environment broadcasting has taken place by this stage all the processing nodes contain the same information; however from this point on the information in the different groups will be determined by their individual master. Each new master copies a control frame from the control memory and proceeds with its standard Prolog execution. If its number of locally spawned OR processes exceeds k, it performs the same operation of partitioning its offspring into masters and slaves. Initially there will be many idle processing nodes but as the number of OR processes increases the partitioning will expand to such an extent that jobs are pushed down the hierarchy until there are only master nodes. At this stage the situation can arise that a master can have more than k jobs to run and there are idle processing nodes elsewhere in the system. If there are more than m idle nodes the implementation allows the state of the overloaded processing node to be copied into all of the idle nodes using the broadcast network. The partitioning mechanism is then restarted.

The best combination of values for the parameters k, g and m will vary with different programs, the balance to be maintained is that of copying overheads (either of frames from the control memory or whole environments in the later stages) and better load balancing between processing elements. Examples are quoted with k = 6, ie a processing node will store locally up to six OR processes before distributing the work. It is not clear whether this is intended to be a realistic value. The guidance as to the determination of g is obtained from analysis of Prolog programs: it is suggested that a value of 0.2 - 0.4 of the average number of untried branches at a choice point, ie if the average number of OR processes spawned at each choice point is 10, the number of processing element designated groups should be in the range 2 to 4 giving an approximate theoretical allocation ratio of three processes to one master processor at the time of splitting into subgroups.

In this system the work load on any processing node throughout program execution is determined by the values k, m and g. Details of tasks awaiting allocation are held centrally and processing nodes are responsible for "collecting" new work as they become idle. It will be shown in Chapters 5 and 6 that the method used to implement a parallel system for the Pure Logic Language has certain similarities in that it recognises the crucial role that broadcasting of environments can play but the allocation of work throughout the machine is implemented in a different manner.

The Delphi project is based on the OR parallel execution of pure Prolog programs [Alshawi 88]. The fundamental approach is that communication overheads can be reduced if separate processes perform a certain amount of recomputation. Under many circumstances it may be speedier to reproduce the parental environment by recomputing it than by passing it in message form across a communication network. The Delphi project has explored ways in which the recomputation of data necessary for OR processes to run on separate processing elements can be employed, and inter-processor communication kept to a minimum.

Conceptually each path through the solution tree has a processor allocated to it and the path is executed in standard sequential fashion. Because no alternatives are represented, no backtracking is involved. Each processor holds a copy of the program, ie the rule base, and is given a code for the path to follow which indicates the branch to take at each choice point. In the solution tree shown in Fig.3.6 the eight paths through the tree have path specifications or "oracles" "111", "112", "113", "121", "131", "132", "21", "22". Given such an oracle a processing node can arrive at its leaf of the tree totally independently from the others.

The naive implementation of this builds up the oracles level by level. The root node on discovery that there are two branches creates two oracles "1" and "2", and despatches them to two processing elements. The processing element receiving "1" recomputes the solution tree taking the first branch until it reaches the three nodes on the next level. The oracles "11", "12", "13" are sent to other processing nodes, each of which start again at the root node and follow their individual path to evaluate the goal list. This method of using oracles to communicate the state of the computation to remote processing elements clearly cuts down on message passing overheads, although the size of the oracle will increase as the tree grows.

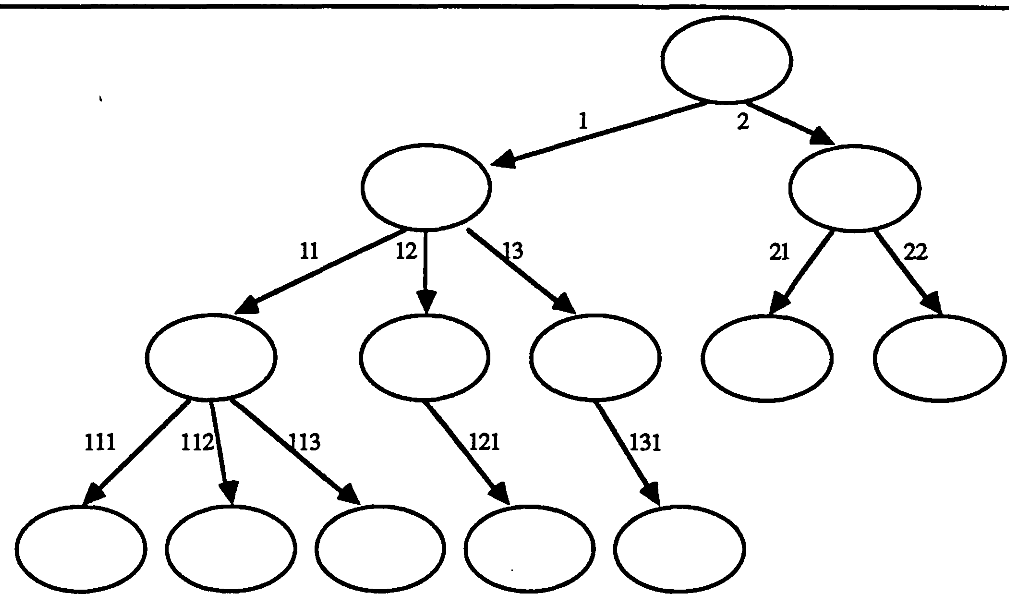


Fig. 3.6 - Oracle Representation

The amount of recomputation involved also increases and can be related to the shape of the tree. A short bushy tree will involve less recomputation than a narrow deep one.

The practical implementation of the oracle model involves the introduction of "bounded depth" backtracking. A process follows the oracle it receives but does not immediately create oracles when alternative nodes are encountered. Instead it uses the conventional backtracking method of handling choice points until some predetermined limit or bounded depth is reached. When this boundary is reached the processor assembles the corresponding oracle or oracles which are then sent to idle processors by a controller mechanism.

The research at present has concentrated on the different ways of defining the bounded depth for oracle creation. Because the potential amount of recomputation is dependent on the shape of the solution tree, it would be ideal if some method of reflecting the nature of the tree could be incorporated into the computational model and allow the system to deal automatically with the different types of search space invoked by various programs. The length of the oracle can be used to give an approximate measure of the amount of potential recomputation involved and thus the oracle size can be used as a means of setting the bounded depth limit of backtracking. This would mean that the bounded depth varied throughout program execution. The work on this approach is still underway and it is not yet possible to make a final assessment of the schemes for optimising the amount of parallel processing involved. However it represents one end of the spectrum of methods of making information available to a number of processing elements and the proposal to be put forward for the parallel Pure Logic Language system incorporates a version of path following by means of a simplified oracle type identifier (see Chapter 6.4.3).

3.2. Architectural Proposals for Multiprocessor Machines

3.2.1. Introduction

The use of large scale multiprocessor architectures in the field of scientific computation has been well established and there are a number of successful commercial systems including the supercomputer architectures [Hwang 85]. However the type of computational demands made by knowledge based systems are very different from the more regular patterns involved in number crunching operations. The aim of this section is to relate the computational demands made by these systems to the design of parallel architectures, and look at a number of examples which display the different technological possibilities. There is no intention to provide a full scale review of multiprocessor machines as many systems are inappropriate for parallel logic languages [Jelly 87].

3.2.2. Computational Requirements for Parallel Architectures

This section analyses the processing requirements for knowledge based systems in relation to multiprocessor architectures.

The broad field of symbolic processing includes all the aspects of knowledge representation looked at in Chapter 2.2 and produces a diversity of applications, eg relational databases, expert systems, logic language programs. The fundamental computational task in all these systems is that of search and its related pattern matching operations. This process has been seen clearly in the logic language implementations but is involved in any systems using some form of chaining inferencing mechanism, such as resolution in logic languages, backward/forward chaining in production systems or graph propagation in semantic networks. When the computational demands made by the search process are looked at it becomes clear that they are very different from those involved in numerical calculations. The actual complexity of the "atomic" pattern matching operation is not great and the amount of communication or message passing is considerable. Thus the granularity of the system, ie the processing/communication ratio, is likely to pose problems. If on the other hand, the notion of the atomic operation is upgraded to involve a series of pattern matching tasks as in the execution of a logic language OR process, the granularity of the system may be improved but the amount of computation involved in each atomic operation becomes very variable, and it is not possible to design the architecture on the basis that one atomic process will execute in unit time.

Memory management for parallel knowledge based systems is also problematic: large memory requirements are needed in these systems, but patterns of computation mean that different processes may frequently require access to the same data at the same stage in the processing. The third aspect which causes difficulties arises from the non determinism involved in knowledge based programs. Because the pattern of processing is not known at the time of querying the system static mapping of computational tasks to processing elements is likely to lead to highly inefficient systems, and some form of dynamic load balancing is necessary to exploit the benefits of parallel hardware.

3.2.3. Design Methodology for Multiprocessor Architectures

The organisation of this chapter and subsequent chapters concerned with the parallel Pure Logic Language system, indicates that the design methodology involved in this project has been top down. In this approach the designer works through the stages of identification of the applications area and its translation into an abstract model of the task. A suitable executable language is then chosen and a computational model for its parallel implementation developed. Finally the design of a machine which will allow as direct a mapping of the computational model as possible is proposed. This is the approach that most parallel logic language systems have employed, and it is noticeable that many of the schemes stop short at major technical proposals for novel parallel machine implementation. The testing of the parallel behaviour of such systems has been achieved either by simulation of the process model and the architecture or by use of an existing architecture which may not exhibit the ideal characteristics for the system, eg the Transputer test bed for the implementation of the BRAVE language [Reynolds 87b].

The other design approach is to start with the multiprocessor architecture and base the computational model of the language on the operations that the machine can handle efficiently. By employing a parallel architecture which appears to possess suitable characteristics for a given form of symbolic processing, the exercise of mapping a language and its computational model may not provide the maximum performance benefits of the top down approach but is likely to give useful comparative information on various aspects of the machine's behaviour. This approach is seen in the study of retrieval of free text documents using the Connection Machine [Stanfill 86]. This multiprocessor machine was originally designed for general fine grained artificial intelligence applications especially the graph traversals in semantic networks but can be employed for a range of applications if they can be expressed in "data parallel" algorithms [Hillis 86]. The point of interest in the document retrieval programs has been revealed by further analysis of the performance of the system: although the multiprocessor machine provides substantial speedups when compared to the theoretical performance of a single processor system using the same programs, when different algorithms are used for the single processor system, the advantages due to parallel processing may be negated [Stone 87].

3.2.4. Shared versus Non Shared Memory Architectures

3.2.4.1. Introduction

The discussion on the requirements for a parallel computer for logic language systems has identified two aspects that are of crucial importance: memory management and load balancing between different processing elements. The relationship of memory to processing elements also impacts upon the communications pattern as non shared memory machines must cater for a different form of communication overheads. Thus the granularity of the system is defined by the computational model and the technological aspects of the target architecture. It would therefore seem sensible that the decision about the relationship of memory to processing elements is taken at an early stage in the design of the computational model. For most of the language systems that have been considered this has been the case, ie they have been designed with either shared or non shared memory systems in mind. It is true for the parallel Pure Logic Language system: this project has taken the view from the outset that for reasons of scalability, large multiprocessor architectures should be based on distributed memory designs, and the computational model of the Pure Logic Language has been proposed to implement a message passing mechanism (see Chapter 5).

3.2.4.2. Shared Memory Systems

In a parallel logic language system the amount of data that needs to be made common to a number of different processes is considerable, and thus many of the proposals have specified a shared memory multiprocessor architecture. These proposals include the Gigalips Prolog systems and the BRAVE implementation. The PEPSys system claims to be machine configuration independent but appears to be using a modified form of global addressing which indicates that some form of shared memory is likely to be used and the proposals for the "multicluster" architecture would bear this out [Chassin de Kergommeaux 89].

The problem with shared memory is the contention of access to memory which leads to the scalability problem. The contention for memory access depends on two factors: the communications network which can range from a common bus as used in a machine such as the Sequent

Balance, to complex multipath high bandwidth systems, and the partitioning of data into different memory segments. Whereas efficient systems containing small numbers of processing elements have been implemented, there is doubt about the feasibility of systems with hundreds (or thousands) of processing elements. Multiprocessor systems have been designed with large numbers of memory banks connected to processing elements with multipath connection networks: the BBN Butterfly has 256 processing elements connected to memory modules by a multilevel Banyan switching network, and ALICE has a similar number of processing elements connected to shared memory by a series of crossbar switches [Rettberg 86], [Harrison 86], [Darlington 87]. However the cost of supporting this versatile multipath connection scheme is speed: in the BBN Butterfly access to remote memory takes about 6 microsecs. This is acceptable for the type of applications such as image processing, VLSI simulation etc, where processing operations may be complex but is not suitable for the memory access requirements of logic languages systems.

Even where contention for the communication medium can be reduced, the viability of the system may depend on the manner in which data can be spread around a number of different memory segments. The case is made for machines such as the BBN Butterfly that memory partitioning schemes can be developed to allow minimum contention in applications such as matrix multiplication, image processing etc, and performance efficiency of up to 90% is quoted to prove this. Unfortunately the patterns of data access in logic languages differ from those involved in numerical applications and make it difficult to install a partitioning scheme that will minimise simultaneous requests for the same memory segment. The introduction of local caches for each processing element can be of assistance as typically dynamic data is written once during the course of logic program execution and thereafter only read [Haridi 89]. This is true of systems which use a hierarchical organisation of binding windows to enable binding values to be "shared" between different processes (see Chapter 3.1.3.2).

The long term view is that shared memory architectures will not provide the optimal vehicle for high performance parallel logic language systems: however the present emphasis on proposals based on shared memory has been encouraged by the recent availability of commercial systems of this type. Machines such as the Sequent Balance and the Encore

Multimax can hold up to sixteen processing elements with access to a large shared memory. The PEPSys system has been implemented on a machine of this sort and results for the Gigalips project on an Encore Multimax are also available [Chassin de Kergommeaux 89], [Lusk 88]. The table below (Fig.3.7) presents sample data from that project, and it can be seen that good speedups relative to the sixteen processing elements have been achieved for some of the applications.

The processing granularity in this system is controlled by the use of public and private sectors as has been described in Chapter 3.1.3.2. The results shows that for a relatively coarse grained system the overheads of using a shared memory with a small number of processing elements are not a problem, making this an attractive approach for small/medium sized commercial applications. However the indications are that there is a much larger potential for parallel execution with these logic language systems which could be exploited given suitable machines with many more processing elements.

Program	2PEs	4 PEs	8 PEs	16 PEs
8Queens	1.98	3.89	7.53	12.4
Tina	1.97	3.84	7.22	11.3
db5	1.88	3.38	5.62	6.35
parse	1.87	3.42	5.28	5.83

Fig. 3.7 - Aurora Prolog / Encore Multimax Speedups

3.2.4.3. "Intermediate" System Proposals

The move to a non shared memory system has resulted in two "intermediate" proposals: the "multicluster" architecture of ECRC for the PEPSys system, and the Data Diffusion Machine for the Gigalips language systems. These are both at the design stage at present so no performance data exists for either of them.

The Data Diffusion Machine proposals represent an attempt to implement a global memory computational model on a fully distributed memory machine [Warren 88a], [Haridi 89]. The system operates on a virtual memory basis in that the location of a data item is decoupled from its virtual address. This will allow the shared memory Prolog schemes developed during the Gigalips program to be implemented directly on the machine, leaving the question of memory management to the machine control elements.

The design allows the actual location of data to be adjusted during query evaluation: this may involve moving the data to a different address or copying it, if multiple copies are needed for easier access. This latter operation is encouraged by the fact that much of the dynamic data in a logic program is written once and thereafter accesses are for reading purposes only. The translation of virtual address into physical address is then the responsibility of the memory control units that are spread throughout the machine, and these each have access to a local directory. The configuration of the machine is a hierarchy of processing units and busses, each subsystem having a directory and a controller to locate and access data. Each controller has two functions: it manages access to data within a given subsystem, and it passes a request for non local data up the hierarchy until a higher level controller recognises that it has jurisdiction over the data. Warren believes that most data accesses can be kept local, ie within a basic subsystem. This design has the advantages of scalability without loosing the practical advantages that a shared memory model of logic language execution provide. However it will require the efficient and frequent transfer of small amounts of data throughout the machine and it remains to be seen if the technology can provide this.

3.2.4.4. Non Shared Memory Systems

The final division of multiprocessor architecture is often separated by the term "multicomputer" to indicate that each processing element holds its own local memory and functions in a more or less autonomous fashion. However this term is not used here: as the previous description of the Data Diffusion Machine has shown there is a continuum of memory configuration schemes from shared to non shared versions, and thus the term "multiprocessor" is retained for all parallel architectures.

The issues involved in the design of suitable non shared memory machines for parallel logic language systems revolve round the communication paths between processing elements. Contention for shared memory is replaced by the need to access data held on distant processing elements, and the ease with which this can be achieved is dictated by the type of communication links between processing elements and the amount of locality that can be incorporated into allocation of tasks.

The connection network that exists in distributed memory machines can be implemented by a static or fixed grid of connections or by a reconfigurable system of switches. There has been a considerable amount of recent work on developing machines based on the hypercube configuration as this form of connection network allows for communication between nearest neighbours and more distant processing elements. This class of machines includes the Connection Machine, a fine grained, centrally synchronised computer with up to 64,000 individual processing nodes [Hillis 85]. However the fine granularity and synchronisation of computation does not provide the necessary functionality for a process based parallel logic language system. Architectures such as the Parsifal system also provide a statically connected communications network at run time [Capon 86], [Hughes 86]. In this machine rows of Transputers are connected together by means of crossbar switches allowing the pattern of connections to be altered for different applications. In general the pattern is set for a particular application run and is not dynamically reconfigured during program execution although recent work has explored the possibility of adjusting the configuration at runtime [Avramov 90].

It has been seen that the pattern of logic language processing which follows the search tree is not predictable in advance and therefore static mapping of processes to processor cannot be considered. This makes the question of locality of processing more difficult. A good system of load balancing will place work on whichever processing elements are least busy regardless of their position in the machine. However if it is required that offspring processes are to be closely connected to their parents, ie on nearby processing elements, in order to make data transfer easier, division of work may not be optimal. Obviously in hypercube architectures where links exits between more distant processing elements as well as nearest neighbours, a more flexible pattern of data transfer will ease this situation.

The ease with which data can be sent between different processing elements is one factor in the design of an appropriate machine for parallel logic languages. However data transfer patterns in these systems display a further characteristic: frequently the need arises for the passing of data from a parent process to several offspring simultaneously. The data to be passed represents the parental binding environment and is thus the same for each offspring. This broadcasting requirement cannot be met on a grid or hypercube form of architecture.

The idealised parallel logic language distributed memory machine thus incorporates a broadcasting mechanism and dynamic reconfiguration of processing element connections in order to allow processing elements holding parent processes to send data directly and simultaneously to a number of offspring processes set up on separate processing elements. The BC-Machine system of SICS aims to meet these two functional requirements, as does the architecture proposed for the Pure Logic Language (see Chapter 6).

The broadcasting mechanism employed by the OR Prolog BC-Machine system has been discussed from the functional point of view in Chapter 3.2.4. The proposed hardware implementation involves the use of crossbar switches to provide the broadcasting operations [Ali 88a]. A two tier system of local crossbar switches linked to a global crossbar switch is described. The design optimises the divisions into "masters" and "slaves" to involve communications through local switches where a processor has a link with every other processor. The crossbar switches can be used because of the nature of the software configuration: at any time during program execution a processing element is either an independent master or it is a slave tied to one master. Thus a processing element is only required to receive information from one other processing element at any given time. This contrasts with the proposals for the parallel Pure Logic language system: in the proposed architecture it will be seen that a processing element needs to be capable of simultaneously receiving data from a number of different broadcast channels because there is no hierarchical division into masters and slaves, and broadcasting can involve the full set of processing elements as receivers (see Chapter 6).

3.3. Summary

The manner in which various logic languages systems have used the concept of parallel execution has been discussed. A number of categories have been identified: some systems implement OR parallelism only whereas others allow different forms of AND parallelism. The factors which affect the design of multiprocessor architectures for use with such systems have been looked at. It has been the aim of this and the previous chapter to provide justification for the decision to concentrate on the implementation of an OR parallel process model for the Pure Logic Language which could be mapped onto a novel distributed memory architecture. Chapter 4 presents the sequential version of the Pure Logic Language and is followed in Chapter 5 by a description of the computational model developed for the parallel execution of the language.

Chapter Four

The Pure Logic Language

4.1. Introduction

The Pure Logic Language (PLL) has been developed at ICL in their Systems Strategy Centre. The work arose out of ICL's interest in large business database systems; in 1986 Babb, the Project Manager, wrote that "the construction of large future information systems will depend increasingly on rules formalised in logic rather than ad hoc algorithms" [Babb 86a]. The vital component in these systems was viewed as the logic interpreter which had to perform in a number of different ways: it had to act as a theorem prover for correct transformation of expressions and provide true reversibility. It must contain trapping mechanisms for expressions which cannot be transformed, theorems to equate equivalent expressions and explanation facilities for the user. Additionally it must be able to incorporate algorithms to allow the actual problems to be solved in an efficient manner [Babb 86a].

It was felt that Prolog based systems were unlikely to meet these objectives because of several "non logical" features. These were identified as order sensitivity, uncontrollable looping, obscure semantics and non standard negation [Babb 86b], [Nairn 87]. Because of this work on a logic system at ICL has taken the form of defining a new language and designing an interpreter for it. This language has formed the basis for the work on the parallel logic language implementation which is documented in this thesis. Before the parallel system design can be discussed it is necessary to describe the important features of the sequential system.

This chapter looks at the early development of the PLL system and discusses the method of inferencing which involves the use of rewrite rules. The version of the language used in this project is defined and the manner in which the sequential interpreter uses the concept of rule rewriting is described. As the parallel version has evolved from this system, a detailed account of the interpreter is presented and this is related to the Warren Abstract Machine implementation of Prolog which has been looked at in Chapter 2.2.6. The PLL system has been documented in various papers prepared by the Logic Language Research Group at SSC, ICL [Babb 83], [Babb

86a], [Babb 86b], [Babb 87], [Babb 89], [Nairn 87], [Cooper 87a], [Cooper 87b], [Cooper 87c], [McBrien 88a], [McBrien 88b], and in [Jelly 88].

4.2. Development of the Pure Logic Language

Early versions of the logic language system were based on the Prolog resolution approach but incorporated the concept known as the "Finite Computation Principle". Under this principle all basic predicates must trap and flag expressions which cannot be reduced to TRUE or FALSE: where this occurs, various standard axioms may be applied in order to allow further transformations to be made. For example the query
(less(x 5) and (x=4))
cannot be successfully evaluated under a left to right resolution based Prolog system as the first subexpression will produce an infinite number of bindings for x. The Finite Computation Principle would trap this infinite loop and by using the logical axiom of commutivity, ie B & A = A & B, reverse the order of the subexpressions and return the result TRUE with x bound to the value 4.

The earlier versions of the language interpreter were written in LISP and maintained the resolution plus Finite Computation Principle approach [Babb 83], [Babb 86a]. However more recent work has moved to an implementation based on the technique of rule rewriting and it is this system that is considered here [Nairn 87].

4.3. Rewrite Rules

The application of resolution based methods to the execution of logic programs is well accepted and there is no shortage of reference to the theoretical basis for them in the literature. However the concept of employing rewrite rules to perform this form of computation is less well established, and discussion on the theoretical issues involved is not readily available in an accessible form.

The use of rewrite rules is more familiar in the context of general mathematics where the successive transformation of an expression into another until some final form is reached is used regularly as a method of mathematical proof. Bundy gives a more formal definition to the concept of rewrite rules in his book on the Computer Modelling of Mathematical

Reasoning and proceeds to show how this approach offers certain advantages over other uniform proof procedures such as resolution [Bundy 83].

Rewrite rules are sets of ordered pairs in which a typical example can be represented as

lhs => rhs

In order to apply the rules a "rewriting rule of inference" is required. This can be defined by the application of the rule

lhs => rhs

to the expression

exp[sub]

where sub represents some subexpression of exp. The application of the rule will result in

exp[rhsø]

where ø is the most general substitution such that

lhsø=sub

The relationship between the lhs and rhs may be equality, inequality, implication, double implication etc.

In general there is likely to be a choice of which rule is to be applied at each step in the proof procedure and a number of different heuristics can be produced for determining the choice. However as the following section will show the rules for the execution of PLL programs are so defined that at all times there is one and only one rewrite rule that is applicable.

4.4. The Pure Logic Language

The version of the Pure Logic Language that has formed the basis of this project was produced in 1988 and is documented in PLL User Guide, Version 0.2 Issue A [McBrien 88a]. A formal BNF definition of the syntax is contained in the User Guide and is reproduced in Appendix B. There have been further extensions to the language since that date but its fundamental nature has remained the same.

The intention is that the language should provide an executable form of "pure" first order logic. Conceptually the system holds a collection of rewrite rules expressed in a syntax similar to that of standard predicate logic, and these rules can be applied to any expression that is entered into the

system in order to "rewrite" it into another expression. Thus if the system holds a rewrite rule concerned with the concept of motherhood,
mother(x y) => parent(x y) and female(x),
when the expression
mother(x y)?
is put to the system it will be converted into
parent(x y) and female(x).
The "=>" notation is used to indicate that this involves the application of a rewrite rule and is not a logical implication as in Prolog.

Rules held in the system are either entered by the user or predefined for the system. In the example in the last paragraph at some earlier stage the rule for "mother" would have been inserted using the syntax:
define mother(x y) tobe parent(x y) and female(x)?
The "?" acts as a terminator and is present in both rule definition and query entries. Inbuilt or system rules are defined for the handling of conjoined and disjoined expressions, equality and negation as well as a number of arithmetic and list processing operations. Existential quantification of variables is included in the language definition. The manner in which these system rules operate and how they are implemented is considered in more detail in Chapter 4.5.

The full syntax for the Pure Logic Language is given in the BNF language definition in Appendix B, and examples of programs, ie collections of user defined rules, are given in Appendix C.

4.5. The Interpreter

4.5.1. Rule Rewriting

The basic philosophy behind the language system is that expressions should be reduced to a minimum expression or fixed point [Cooper 87b]. This can take the following forms:
a) FALSE,
b) TRUE,
c) TRUE with variable bindings,
d) TRUE with a set of alternative variable bindings,

e) the most simplified or reduced form of the query, eg the query
(less(x y) and (y=3))
would reduce to
(less(x 3) and (y=3)).

The method that the interpreter uses to achieve this is the technique of rule rewriting. Essentially the interpreter holds a set of rewrite rules. When an expression is put to the system for evaluation the interpreter successively identifies and applies the appropriate rewrite rules until the expression cannot be reduced further and has reached its fixed point [Nairn 87]. This evaluation process is divided into two steps: the initial parsing of the input expression into a form that is recognisable to the rewrite manager, and the subsequent operation of the rewrite manager in successively applying appropriate rewrite rules.

The parser operates by converting the input expression into an expression tree, eg the expression
a(x) and b(x) and (x=9)?
would be transformed into the tree shown in Fig.4.1. The detailed data

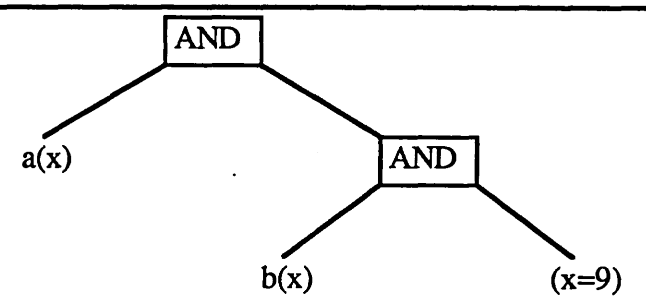


Fig. 4.1 - AND Node Expression Tree

representations used in the creation of expression trees are looked at in Chapter 4.6.3. The tree having been defined by the parser, control is handed to the rewrite manager which determines the appropriate rewrite rule by reference to the root node of the expression tree and then applies it. In this example the first rewrite rule to be applied is the inbuilt rule for conjunction because of the AND node at the root of the tree.

In outline therefore the PLL rewrite system works by the application of the following process:

```
{read input string,
 call parser to convert string to expression tree,
 call rewrite manager,
 while ((tree_root_node) ≠ TRUE_NODE or FALSE_NODE)
                    and (expression not "uncomputable")
  {case (tree_root_node)
     AND      : call rule for conjunction evaluation,
     OR       : call rule for disjunction evaluation,
     EQUALS   : call rule for equality evaluation,
     TIMES    : call rule for multiplication evaluation,
     SQRT     : call rule for square root evaluation,
        ..    ..   ..
     RULE     : call appropriate user defined rule.
  }
 endwhile,
 return(tree_root_node).
}
```

The final choice in the case statement represents the call to evaluate a user defined rule: all previous options refer to inbuilt system rules. It can be seen at this level that the functionality of the system can be extended if required by the inclusion of new system rewrite rules. For example if it were desirable to include trigonometric function evaluation in the system, the interpreter could be modified to allow the parser to produce sine, cosine, etc nodes in the expression tree. Rules for the evaluation of the appropriate trigonometric function could then added to the above list. (In later versions of the PLL system these rules have been implemented [McBrien 88b]).

The interpreter can thus be viewed as consisting of a collection of rewrite rules [Cooper 87b]. These are of two types: system or inbuilt rules, and user defined rules. User defined rules are the equivalent of a program in a conventional language system and may include any of the connectives and functions shown in the formal syntax definition. Recursive definitions are permitted as in Prolog, but the rules cannot be altered dynamically at run time, ie there is no concept of "asserting" or "retracting" part of the rule base while a query is being evaluated.

The system rewrite rules provide the mechanism for the evaluation of the logical connectives (and, or, not), the arithmetic and list processing operations. These rules are applied to the query expression in a predetermined fashion and it is this combination of ordering of rule rewriting plus the actual effect of the rule that provides the correctness within the system.

4.5.2. AND Node Rewriting

In Chapter 4.2 it has been shown how the Prolog approach fails with the query
(less(x 5) and (x=4))
because of the ordering of the two subexpressions. When this query is put to the PLL rewrite rules the first rule to be invoked is the meta rule for conjunction rewriting. This rule works by evaluating the left subexpression (or left branch of the expression tree) but in the event of this being uncomputable, ie irreducible to TRUE or FALSE, it passes to the second branch to evaluate it. If bindings are made on this second rewriting, the conjunction evaluation algorithm returns to the first branch to test if the variable bindings will influence its rewriting. By applying this method the whole expression is reducible to TRUE with x bound to 4. This method of conjunction evaluation is recursive and will apply to any number of conjoined expressions. Its application means that the problems of order sensitivity associated with Prolog are overcome in the PLL.

This description of the operation of conjunction rewriting rule shows that the usefulness of the interpreter depends on the correct algorithms being available for each rewrite rule. The algorithms used to implement the basic logical operations are based on accepted logical axioms: de Morgans laws, laws of commutivity and association, etc. [Cooper 87a], [Cooper 87b], [Nairn 87].

4.5.3. OR Node Rewriting

Disjunctions, ie "OR" expressions, are rewritten by evaluating each branch or subexpression separately. If every subexpression is computable the disjunction will return return FALSE or TRUE with a set of alternative bindings. If disjunction is nested within a larger expression, eg
(p and q)

where p and q are logical expressions and p is rewritable to
(p1 or p2 or p3)
the whole expression will be transformed to
((p1 and q) or (p2 and q) or (p3 and q)),
and each subexpression is then subject to evaluation under its appropriate rewrite rules. The manner of implementation of this involves the storage of the environment for the alternative branches of the disjunction: this is described in the section on the implementation of the interpreter.

4.5.4. IN Node Rewriting

Related to the disjunction rewrite rule is that for membership. List membership is indicated by the use of the "in" predicate, eg
define female(x) tobe [x] in [["sarah"] ["betty"] ["frances"]]?
This predicate can be used to define base predicates or ground clauses, the above definition corresponding to the Prolog
female(sarah).
female(betty).
female(frances).

The rewrite rule that handles membership transforms the "in" predicate into disjunctions of equality. Thus a query to a system containing the above rule definition
female(x)?
would result in the nested disjunction
((x="sarah") or (x="betty") or (x="frances")).
The disjunction rewrite rule plus the equality rewrite rule would further reduce this to
TRUE with the set of alternative bindings for x, ie "sarah", "betty" and "frances".

4.5.5. NOT Node Rewriting

Unlike Prolog the PLL implements negation correctly. This is defined as "classical" negation, rather than negation by failure [Cooper 87b]. The rewrite rule implements the usual negation manipulation rules of predicate calculus, ie de Morgans laws, elimination of the double negation. Where a double negation of an expression containing free variables the PLL

rewrite rule will give the "correct" evaluation, ie rules for p and q have been defined as:
define p(x) tobe (x="a")?
define q(x) tobe (x="b")?
the queries
((p(x) and q(x))?
and ((not(not(p(x)))) and q(x))?
will both respond FALSE.
In Prolog the equivalent second query would succeed with x bound to "b".

In the PLL where negation is encountered at the outer level in a conjoined or disjoined expression, eg
(not(p and q))
the expression is rewritten using de Morgans laws to
(not(p) or (not(q)).
Similarly
(not(p or q))
is rewritten to
(not(p) and not(q)).
Negation is thus moved inwards to be applied to the logical expression at its most reduced level.

4.5.6. User Defined Rule Rewriting

The rewriting of user defined rules involves the replacement of the left hand side of a rule with the right hand side. When a query referring to a predicate name is put to the system, the predicate name is matched against the list of user defined rules, and if a rule exists for that predicate the substitution is performed with appropriate variable unifications.

The PLL does not allow constants to appear in the variable list of predicates; the Prolog rules which state that Bill likes anyone who plays football and gives two examples of games players are:
likes(bill, X) :- plays(X, football).
plays(sam, football).
plays(fred, tennis).
These are defined in the PLL as:
define likes(x y) tobe (x="bill") and (some(game)(plays(y game) and
(game="football")))?

define plays(x y) tobe [x y] in [["sam" "football"] ["fred" "tennis"]]?
This means that the process of unification of variables is a simple pattern matching operation which maps the user's variable names onto the internal variable representation held with the user defined rule. If the query
likes(x y)?
is put to the above set of PLL rules the response will be
(x="bill") and (y="sam"),
similarly the query
likes(x y) and (y="fred")?
will be answered with FALSE.

4.5.7. Future Optimisation of Rewrite Execution

In the same way that various optimisations have been incorporated into the implementation of Prolog systems it is envisaged that the algorithms used to implement the rules can be made more efficient before being applied to a commercial system. Indeed the concept of the interpreter as a collection of rewrite rules allows for the possibility of rules to be mapped directly onto specialised hardware: in the context of the sequential system and database applications, rewrite rules that involve searching relational tables of data could be directly implemented by using database machinery such as CAFS [Howarth 85]. As the following chapters will describe this project has proposed an alternative method for the rewriting of disjunctions and has mapped this onto a more general parallel multicomputer architecture.

4.6. The Implementation of the Interpreter

4.6.1. Introduction

This section looks at the manner in which the interpreter is implemented: the data structures involved and the overall functionality of the system are discussed. The algorithms which operate the inbuilt rules are looked at and particular attention is given to those for conjunction, disjunction and membership as these are of crucial importance in the move to a process based parallel system.

As discussed in Chapter 4.5, conceptually the interpreter holds a set of rewrite rules which are used to reduce a logical expression to its most basic form. The task of the interpreter is twofold: to determine the order in which rewrite rules are applied and to apply the algorithms which implement the transformations specified for the corresponding rules. It achieves the first objective, namely the ordering of rules, by the method of parsing the incoming query. From that stage onwards the rewrite rules themselves take over the evaluation of the query. It is therefore appropriate to consider the interpreter as performing two different tasks, first the parsing of the query and secondly the rewriting of the query. As will be demonstrated the method of parsing a query is also used for the insertion of user defined rules. However before considering its functioning the overall design of the interpreter and the data structures involved in memory management have to be described.

The interpreter consists of several interactive modules: the main program which controls the system's functioning and holds several general utility functions, the parser, the memory management system which includes user defined rules, and the "core" interpreter or rewrite manager module which contains the algorithms for the inbuilt rewrite rules. There are also two small libraries of mathematical and list processing functions.

The interpreter source code is written in C, and can be compiled to run on the Sun workstation or an Archimedes microcomputer. During this project both systems have been used, although the bulk of the development of the parallel interpreter was done on a networked Sun 3/60 workstation. (Later versions were ported to a Transputer based system - see Chapter 7.4.3.3). The sequential system as developed by ICL comprised six separately compiled modules and occupied approximately 100 Kbytes.

4.6.2. Memory Management Data Structures in the Interpreter

4.6.2.1 The System Stack

The main data structure used in the PLL memory management is the system or evaluation stack. This is a tripartite structure incorporating:

a) the user defined rules area,
b) the query evaluation area,
c) the variable area.

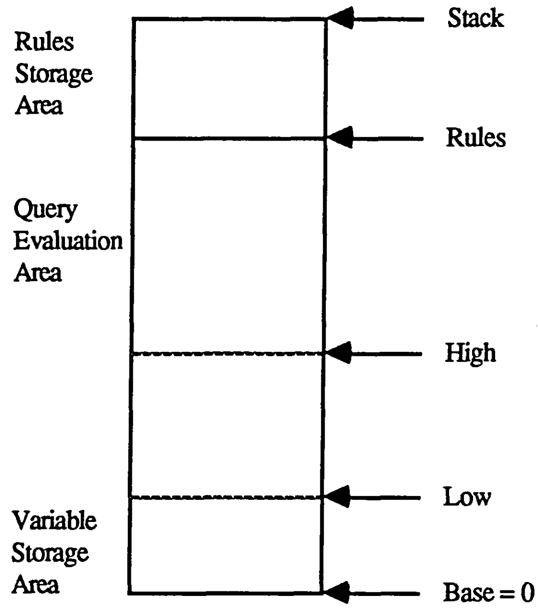


Fig. 4.2 - PLL System Stack

The state of the stack during query evaluation is shown in Fig.4.2. The rules area is differentiated from the rest of the stack by the heavy line showing that during query evaluation no alteration to the rules is allowed, ie no "assert" or "retract" is permitted. This area is used to store user defined rules and only the separate operations of rule insertion or deletion can affect it. The manner in which rules are stored is described below. The space marked as the variable area is used to represent variables that exist during query evaluation. These may be user variables, ie ones which have been introduced in a query, or internally produced ones from rule rewriting. If variables become bound during the rewrite process a value (or a pointer to a value in the case of a list or string) is inserted in the position in the stack designated for the variable.

The query evaluation area holds the internal representation of the state of the query as it is rewritten. This representation is based on a tree structure and is described below in Chapter 4.6.3. As the tree is altered during rewriting it may expand or contract according to the effect of the rules on it. The pointer "High" marks the first free position in the stack below the query tree. Similarly "Low" represents the first free value above the stored variables area. The system will run out of space if during query evaluation "High" and "Low" meet.

The size of the system stack is determined at run time by the C dynamic memory facility. In order to ensure that as much space as possible is provided for the stack the interpreter sets up all the other necessary data structures which are fixed at compile time, and then uses a call to the C function "malloc" to obtain as much memory as is left for the system stack. The value given to the stack in the Sun system is 1,600,000 words and in the Transputer based version 400,000 words, word length being four bytes in each case.

4.6.2.2. The OR Stack.

This is a small array which is used as a stack to store alternative subexpressions resulting from disjunction rewriting. These alternatives are in fact held on the general system stack and the OR stack merely holds the pointers to their position on the main stack. This temporary storage of alternatives represents the list of independent expressions to be evaluated and can be regarded as holding backtrack points in the sequential version.

4.6.2.3. The Variable List

The use of the variable area in the main stack has been described: variables are allocated a two word space on the stack which if the variable becomes instantiated holds the data type of the binding (integer, string etc) and its value (or pointer to the value). Internally variables are referred to by the offset of their stack address from the base of the stack. However from the user's point of view this offset is meaningless, and therefore a structure is needed to link the variable as known to the user with the stack offset. This information is held in the variable list.

The variable list consists of a small array of structures which hold data on the name and type of the variable, the index of the array serving as the connection with the stack base offset. At present the maximum number of variables allowed in a user query is twenty, thus the array consists of twenty elements.

4.6.2.4. The Binding List

When variables are bound during query evaluation the values are inserted in the appropriate position in the stack. However it is necessary to maintain a list of how many variables have been bound and whether they have been bound during the evaluation of any particular subexpression. The binding list provided this information. It is an array holding the stack offsets of any variables that are currently bound and is operated as a stack. The index is referred to as the "binding level" and is used to indicate how many variables are bound at the start of a rewrite operation. Any increase in the value of the binding level would indicate that further bindings have been made. This information is used in two ways: first as has been shown in Chapter 4.5, the conjunction rewriting algorithm relies on this information to determine whether or not it is appropriate to attempt to re-evaluate one of the subexpressions. Secondly when a disjoined subexpression has been fully rewritten, the interpreter uses the binding list information to add a conjoined list of all appropriate bindings to the disjunction. These variables are then subject to "debinding" ie their stack reference is reset to unbound, so that any further disjunction can make new bindings for the variables. In the example query

(a(x y) and (x=8*5)) or (b(x y) and (x=sqrt(4)) and (y=(7+3)))?

the first disjunction to be rewritten will result in a binding for x but not y. This will be noted in the binding list and once the subexpression has been fully evaluated, the expression (x=40) will be conjoined to it, and the stack reference to x changed from bound to unbound. The second disjunction will then be able to install a different value for x in the same stack position during its rewriting and this will be duly noted in the binding list. Thus as evaluation of each disjoined branch of the expression tree is complete the information in the binding list allows the interpreter to add the appropriate bindings to the subexpression.

4.6.3. The Parser

The task of the parser is to convert the incoming query or rule definition into a structure that is stored on the stack, and in the case of a query is then subject to rewriting. The structures which represent rule definitions are stored in the top of the stack as shown in Fig.4.2; the query structure is stored immediately below the bottom of the rules area. The basic format of the structure is similar for rules and queries, the differences being highlighted below.

An incoming query which is linear in format is transformed into an expression tree. The parser creates node structures to build up the links in

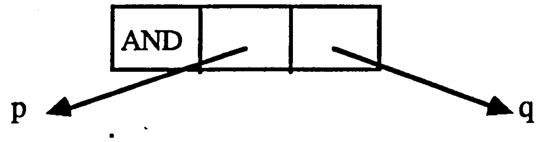


Fig. 4.3 - AND Node Representation

the tree. These can be of two general types: three part nodes representing binary branches, and two part nodes representing a direct link. The "extra" field in the node holds the name of the node, eg the expression (p and q) where p and q are logical expressions is transformed into an AND node which represents the binary tree as shown in Fig.4.3. The second and third fields hold pointers to the nodes for p and q respectively. Nodes are held on the stack and allocated contiguous memory space for each field.

Three part nodes include AND, OR, PLUS, EQUAL, SOME; two part nodes include NOT, the data type nodes NUM , LIST and STRING, and the variable node IDENT. Predicates are given a three part CALL node with the second and third field pointing to the predicate name and its parameter list respectively. This is not meant to be a definitive list of all nodes in the system but indicates some of the building blocks that the parser uses to construct the expression tree. Fig.4.4 shows the expression tree that would result from the query
a(x) and b(x) and ((x=1) or (x=2))?

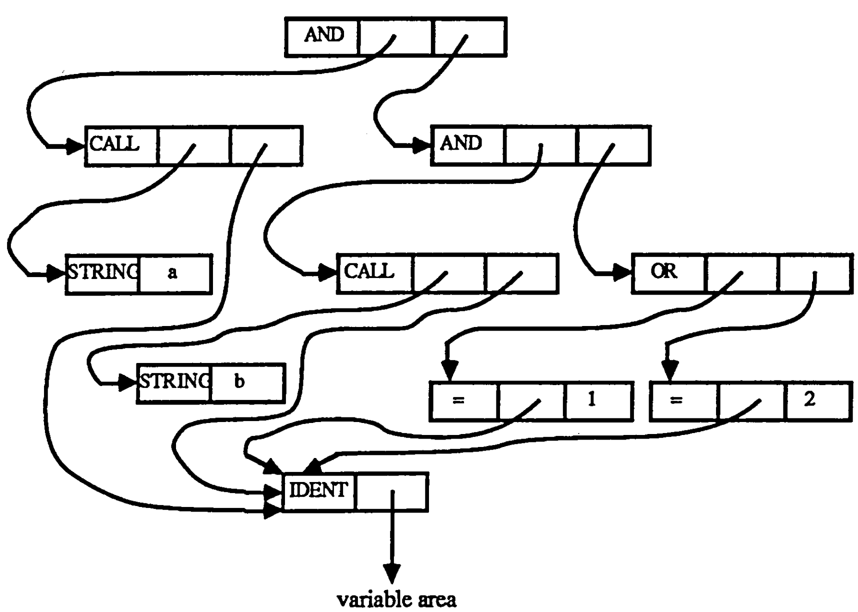


Fig. 4.4 - Expression Tree Representation

When user defined rules are installed in the rules area of the stack they are stored in a similar expression tree. The structure to represent a rule is larger, consisting of fourteen contiguous memory locations. This is because the string holding the "name" of the rule is included in it and rule name can be up to ten characters in length. The rule structure is shown in Fig.4.5. The first two fields hold pointers as shown, the number of variables in the rule head is given in the next field and the number of quantified variables in the rule body in the field no.3. The rule body is represented by an expression tree created in the same fashion as the query tree, eg for the rule

define a(x) tobe b(x) and c(x)?

the rule body will have an AND node as its root and two CALL nodes on each branch. In this instance there are no further rules defining the predicates "a" or "b", and the second fields of the CALL nodes for them point directly to the string identifying them as is shown for the example in Fig.4.4.

0	1	2	3	4 - 13
Rule Body	Next Rule	No. of Vars	No. of Q. Vars	Rule Name

Fig. 4.5 - Rule Representation

However in the event of there being a previously defined rule for "b", the parser will identify this and instead of putting a pointer to the string "b" in the second field of the CALL node it will insert a pointer to the rule structure defining "b". Thus the rule area is built up, not as a list of rules but a network. This "precompilation" of the rule list means that search time is eliminated in the process of query rewriting.

In a similar fashion when a query is entered that contains a reference to a predicate that is defined in the rule base, the parser will identify this and create the appropriate pointer to mark the connection. Thus a minimum of search is involved in query evaluation and it is performed in connection with the parsing operation.

The parser is thus responsible for the setting up of the rule network when user defined rules are inserted, and for creating the initial expression tree for the query. In the same way as it creates the inter-rule links at rule creation time it installs any connections between the query and the rule base at the time of parsing the query.

4.6.4. The "Core" or Rewrite Manager Module

When a query has been parsed and transformed into an expression tree stored in the evaluation area of the stack, control returns to the main program for query rewriting. This is performed by the rewrite manager accessing and executing the appropriate rewrite rules. As has been shown user defined rules are stored in the rules area of the stack, and the inbuilt system rules are contained in the rewrite manager module. This consists of a number of different high level functions which each implement the algorithm for rewriting a particular type of expression. The type of expression to be rewritten is known from the node name on the stack.

The rewriting is effected by starting at the root node of the query and applying the appropriate rule as indicated by the node. Because of the manner in which the meta rules for conjunction and disjunction operate, there is never any choice of which rewrite rule should be next applied. This means that there is no need to search for an appropriate rule. The initial rewriting of a user defined rule substitutes the body for the head of the rule by the process of rule "expansion": this is performed by the copying of the rule body into the query evaluation area of the system stack. This replaces the pointer to the rule head which the initial parsing operation installed. Subsequent rewriting of the rule will operate on this copy and may involve further rule head/rule body substitutions.

As the expression tree is subjected to the transformations as defined in the rules it will expand and contract, the root node being successively replaced by the result of the rewriting. The final tree represents the system's response to the query and ideally is reduced to a pointer to FALSE_NODE or TRUE_NODE, in the latter case with possible binding values attached to the variable representations. In a system that contains the following rule definitions

define a(x) tobe b(x) and (x=8)?

define b(x) tobe c(x)?

the expression tree transformations are shown in Fig.4.6. They involve calling the rewrite rules for AND, CALL and EQUAL nodes. The manner in which conjunction evaluation is operated is described in outline in Chapter 4.5. It can now be seen that it involves a "rewrite left as far as possible, rewrite right as far as possible, then repeat until no more alterations possible" algorithm. Clearly for a query containing many conjoined subexpressions this approach can lead to heavy computational demands. Thus while it is correct to say that the PLL approach eliminates order sensitivity in terms of guaranteeing a computable result where possible, the order in which rules and queries are entered may effect the performance of the system.

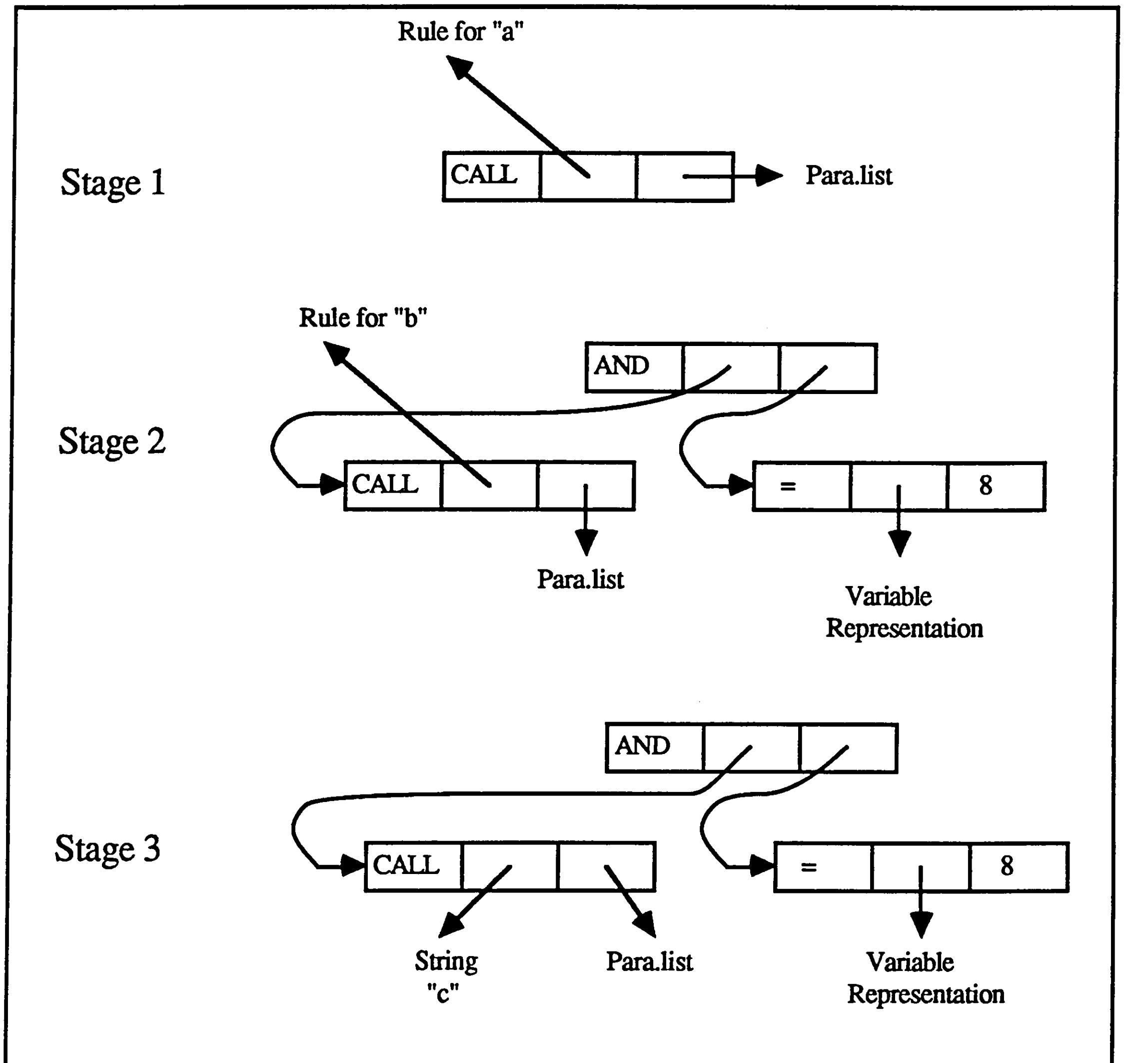


Fig. 4.6 - Expression Tree Transformations

The rewriting of disjunctions and membership is of special interest as these rules represent the handling of alternatives within the system and need to be altered in an OR parallel version. As described previously the OR stack is used to hold a list of alternative branches of the expression tree. The actual expression trees representing the branches remain on the main stack

and the OR stack holds a pointer to the root node of each branch. However because of the binary nature of the tree only two pointers are put on the OR stack at one time, and in many instances one is removed forthwith for independent evaluation. Similarly when the "in" predicate is rewritten one OR node is created, having an equality node as one branch and an altered "in" predicate as the other, eg the query
(x in [1 2 3])?
is rewritten to
(x=1) or (x in [2 3])?
The subsequent OR node rewriting once again puts two pointers on the OR stack.

4.7. Comparison of PLL and Prolog Implementations

The introduction to this chapter has shown that work on the Pure Logic Language was initiated in order to rectify some of the operational problems with Prolog. This section is concerned with how the systems differ in their implementation. The description of the data structures and memory organisation in the PLL given in Chapter 4.6 has made no reference to the similarities that exist between it and the Warren Abstract Machine which forms the basis for the implementation of most current Prolog systems (see Chapter 2.2.6) and it is appropriate at this stage to compare the two systems [Warren 88a].

The initial and most obvious difference is that the PLL system does not hold compiled machine code for the user's program, instead it relies on the parsing operation to produce a form of linked network of rules. As rules are inserted the parser is responsible for creating new connections in the network as appropriate.

When a query is input the parser is again responsible for linking in the query to the rule network (where possible) and thus the search tree is already partially created by the time the rule rewrite phase begins. Rewriting involves operating on this embryo search tree expanding and pruning it, and eventually reducing it to a minimum form. However the rewriting of each of the user defined rules, ie the user's program code, involves the use of generalised matching algorithms not customised compiled code as in the WAM.

In terms of memory management the PLL execution stack can be compared to the local stack on the WAM in that the state of computation is represented by a linked structure: environment frames in the WAM, an expression tree in the PLL interpreter. There are differences in the manner that variable bindings are stored: the WAM is likely to include them in the environment frames whereas the PLL uses a separate data area on the stack and uses a binding list to reference them.

In the PLL alternative expressions are held as expression trees on the system stack but not linked together. Instead a list of pointers to the alternative expression trees yet to be explored is held in the array known as OR stack. However there is no conceptual difference between the processing of alternatives in the two systems.

The crucial difference in the two approaches appears to be the method of handling of conjoined expressions. Because Prolog uses a procedural interpretation of a resolution based inference mechanism, the ordering of which subgoal is to be expanded is fixed at system definition time and in the case of a program with no alternatives follows a deterministic path. This is not the case with the PLL where the method of rewriting conjoined expressions can involve non determinism even when no alternatives are involved. As an example of this, consider the system containing the rules:
a(x) => b(x) and c(x)?
c(x) => (x=9)?
If the query a(x) is put to this rule base the expression tree
(Fig.4.7) will be rewritten in the following order:
1. Rewrite lhs of top AND node ---> return b(x),
2. Rewrite rhs of top AND node ---> return c(x),
3. Rewrite c(x) ---> return (x=9),
4. Rewrite (x=9) ---> bind x to 9, return TRUE,
5. Rewrite lhs of AND node --> return b(x).

If however the rule for c(x) was defined as:
c(x) => d(x)?
step 3 and 4 would be replaced by
3. Rewrite c(x) ---> return d(x),
4. Rewrite d(x) ---> return d(x),

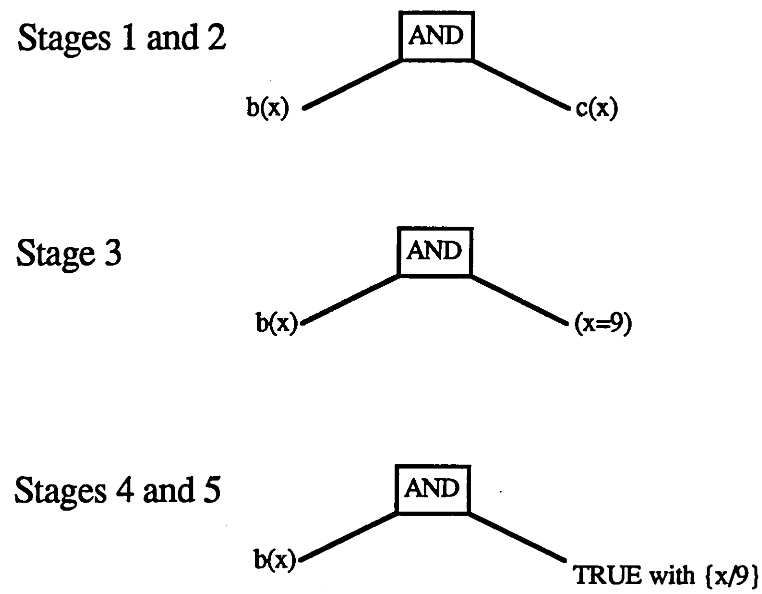


Fig. 4.7 - AND Node Rewrites (Version 1)

and the evaluation would terminate at this stage as step 4 has not produced a binding for x (see Fig.4.8). In other words b(x) is evaluated twice in the first instance and only once in the second.

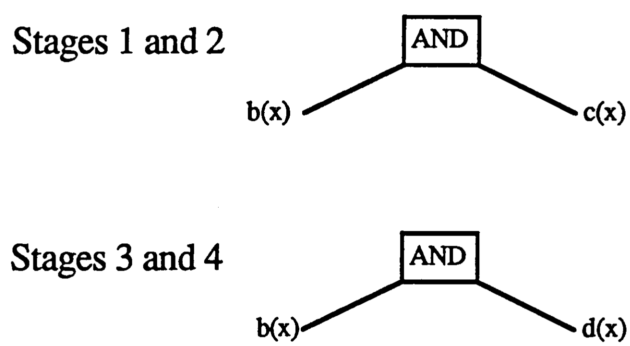


Fig. 4.8 - AND Node Rewrites (Version 2)

This form of unpredictability in the computational path does not exist in Prolog. In the PLL it is the mechanism used to ensure that the ordering of subexpressions does not affect the outcome of expression evaluation but it clearly imposes computational overheads and makes the move towards a fully compiled system for the PLL more problematic.

4.8. Summary

This chapter has described the initiation of work on a new logic language system by ICL. The method of inferencing to be used as a basis for the deductive capacity of the language relies on the concept of rule rewriting, rather than the resolution principle on which Prolog is based. The language has been described and the implementation of the interpreter has been discussed. Particular attention has been given to the meta rules concerning the rewriting of conjunctions and disjunctions as these are important as the move to a parallel system is considered. The PLL interpreter has been discussed in relation to the Warren Abstract Machine, which forms the standard method of implementation for Prolog.

Chapter Five

The Parallel Pure Logic Language

5.1. Introduction

This chapter describes the development of a computational model for an OR parallel Pure Logic Language system and the implementation of the interpreter. This work represents the fusion of the PLL rewrite rules approach and concepts derived from the study of parallelism in logic languages. It has led to the construction of an interpreter which is based on the sequential version written by ICL, but which provides for parallel execution of alternative paths in the solution tree.

The Pure Logic Language contains no execution control structures in its sequential version and it has been a primary aim to maintain this approach when considering the introduction of parallelism. This means that parallel execution is implicit in the system and must be controlled automatically, not by the programmer. This approach separates the project from much of the mainstream work on parallel logic languages as has been discussed in Chapters 2 and 3.

The second premise on which the work on the parallel PLL system is based concerns the applications area. In Chapter 2 it has been seen that the use of logic languages as the programming methodology is particularly appropriate in a number of areas. These include many of the systems designated by the term "artificial intelligence", eg expert systems, natural language programs, knowledge bases. The other area which is intimately related to logic is that of deductive databases. It was interest in this latter area that provided the initial impetus at ICL for work on the PLL. It has therefore seemed appropriate to consider the use of the PLL primarily in the types of application which could be broadly described as knowledge based systems or deductive databases.

With the intention of implementing implicit parallelism and gearing the system towards knowledge base/deductive database systems, the potential for parallelism within the PLL has been looked at. The following section gives this analysis and shows why it has been decided to concentrate on the implementation of OR parallelism in the first instance. The remaining sections in this chapter document the development of the computational model for a process based OR parallel PLL system and the design of its interpreter.

5.2. Parallelism within the Pure Logic Language

The potential for parallel execution within logic programming languages has been described in Chapter 2 with particular reference to the concepts of AND and OR parallelism [Conery 83], [Conery 85], [Hogger 84]. In Chapter 4 the method of execution of PLL programs has been discussed and it can be seen from this that the language includes the notion of conjunction and disjunction of subexpressions [Nairn 87]. Although the rewrite rules for conjunction and disjunction evaluation at present specify a sequential implementation there is no theoretical reason why new rules should not be developed to allow for parallel execution of conjoined or disjoined expressions.

The case for parallel execution of disjoined or conjoined subexpressions can be made if the performance benefits to be gained from this outweighs the computational overheads in setting up the parallel processes and exporting them to distant processing elements. This will depend on the number of candidates for parallel execution and on the architectural features which influence the computational overheads. In other words although the analysis of programs will provide a guide to the value of implementing parallelism, the real performance benefits can only be assessed in terms of a mapping to a particular hardware system.

When the programs are analysed for the potential for OR parallel execution, the prospects look encouraging. Knowledge based systems and deductive databases contain large numbers of alternatives, both in the higher level rules and in the base predicates. These are types of systems which fall into the broad category of "Datalog" programs. Chapter 2.3.5 has presented the analysis of potential OR parallelism made by Ciepielewski for a set of test programs [Ciepielewski 86]. Thus it would appear that the scope for concurrent execution of alternative versions in this type of application is considerable and the benefits to be gained will revolve round the degree to which a system can be defined to minimise the computational overheads.

As discussed in Chapter 4 in the PLL alternatives arise in connection with OR, IN and RANGE nodes and the rules for the rewriting of these nodes will have to be redefined. The effect of the move to an OR parallel basis on other aspects of the rewrite system will be explored in the following sections of this chapter.

The question of the benefit to be gained from the inclusion of a form of AND parallelism is more difficult. As has been shown in Chapters 2 and 3 if parallel execution is to be transparent to the user, it must be automatically generated by the system. This is not so difficult to organise in the case of OR parallelism as alternative branches in the solution tree represent independent computations. However with AND parallel definition the question of the shared variable arises, and some method of determining these dependencies has to be devised. This can take the form of compile time or run time analysis (see Chapter 3.1.2.2). Run time analysis is likely to produce the better result in designating the subexpressions that can be executed in parallel but it inevitably involves computational overheads.

In order to determine whether the potential amount of AND parallel execution is sufficient to warrant the development of a variable dependency scheme, analysis of various programs used by ICL was performed. Of course these programs were developed for use in a sequential system, and it begs the question about programming techniques for a parallel environment, albeit one in which parallelism does not have to be specifically indicated. Appendix D gives the details of this analysis for an example program.

It can be seen from this analysis that the scheduling of AND processes in a manner determined by the variable dependencies will lead to only a limited amount of parallel execution, and this state of affairs was found to be true for many of the test programs developed by ICL. Any program that relies on a recursive rewrite rule definition uses a shared variable as the vehicle for passing data into the next level of the recursive call, and there is no way in which this can be parallelised.

Thus the effort of designing an algorithm to work out variable dependencies for the PLL does not look as if it will provide real benefits at this stage. Active research work in this area is continuing at various centres and future decisions about the inclusion of AND parallelism into the PLL should be made in the light of new results being produced.

5.3. The Parallel Process Model of the PLL

5.3.1. The Requirements of the Model

The fundamental concept behind the definition of OR parallelism is that each alternative branch in the solution tree represents an independent computation. The computational model has to be designed in such a way as to achieve this.

The first step is to consider the granularity or atomic computational unit of the system. This project has taken the notion of a "process" as being the indivisible unit of work. A process consists of a number of computational steps which are defined by the logical demands of the abstract model rather than measurement of computational time or memory usage. This process based approach is well established in the parallel execution of logic programs and provides a sound theoretical basis for the system. However it is worth emphasising at this stage that there are practical problems involved with it: although a process based abstract model allows the language designer to view computation in clear cut terms, the execution of processes will provide a "mixed" granularity system as processes may vary considerably in the amount of actual work involved in each one. This of course gives rise to architectural and scheduling problems.

Having fixed the unit of computational work as a process the requirements of the parallel model are now considered.

The primary requirement of the model is that it should support OR parallelism. Following the discussion in the previous section it was recognised that in the applications area for which the PLL is likely to prove most useful, the simultaneous execution of alternative branches in the solution tree should produce good performance improvements. This gives the rise to the concept of a process based OR parallel system which allows the alternative branches in the solution tree to be defined as separate and concurrently executing processes.

The second requirement for the model is that OR processes should be defined in a manner as to make them genuinely independent of each other and their parent. The expression tree resulting from the query
a(x)?

in a system that contains the rule
define a(x) to be ((b(x) or c(x) or d(x)) and e(x))?
is shown in Fig.5.1.

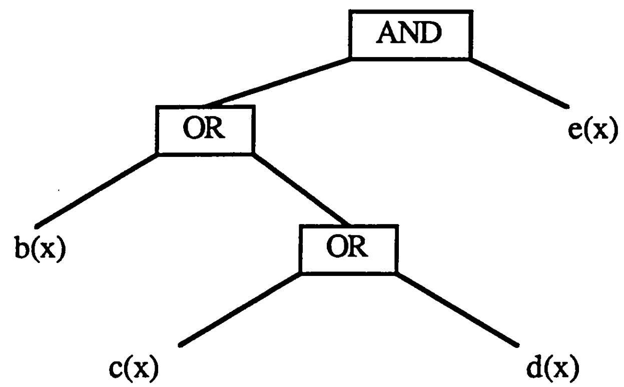


Fig. 5.1 - Expression Tree

One way of regarding the OR parallel execution of the query a(x)? would be to organise the simultaneous evaluation of b(x), c(x) and d(x), and then to report the separate results back to the parent, ie a(x) before the evaluation of e(x) is attempted. This approach involves communication in two directions between parent and offspring, and the descheduling of the parent process while awaiting the results of the child processes. A good description of this type of model is given in [Conery 83].

It was decided to take a somewhat different view of OR parallel execution and in the model of independence defined for the parallel PLL system, the manner of evaluation of the solution tree is the immediate setting up of three processes, ie (c and b), (d and b) and (e and b). These are now fully independent and can run to completion without any scheduling or synchronisation required between them and the parent process. This is the approach taken in the BC Machine project [Ali 88a]. The implications for this approach are discussed in Chapter 5.3.2.

The third aspect of a parallel process model has been referred to in the previous section and is derived from the aim of implementing implicit parallelism. As there is no intention to allow control structures for the designation of parallel execution, the system must be responsible for this, and thus OR processes must be generated automatically from within the interpreter.

In summary the parallel process model for the PLL must
a) support OR parallelism,
b) provide full independence of processes,
c) allow automatic generation of processes.

5.3.2. The Definition of the Computational Model

In order to meet the first requirement as discussed above, ie that of providing OR parallel execution, a process is defined as the flow of computation involved in rewriting an expression and it exists until an alternative branch of the expression tree becomes rewritable. At this point the process spawns offspring processes to correspond with the alternative nodes and terminates. If no alternative nodes are encountered during rewriting a process terminates when it has reduced the expression to its minimum or fixed point (as with the sequential version). Processes can thus be spawning or non spawning, the latter corresponding to the leaves of the solution tree. In a situation where no OR nodes exist the whole query evaluation will take place in one process.

The spawned processes become candidates for parallel execution: whether they are actually evaluated simultaneously will depend on the architectural considerations and the available computational resources but the model provides for the possibility.

The method of process spawning involves message passing between a parent process and its offspring. Essentially when a process encounters an OR node it creates a message structure for each alternative containing the information required to establish the offspring process. These messages are used to trigger the creation of new processes: in a "real" system some or all of these messages would be transmitted across the communication medium to other processing elements to inaugurate the execution of the new processes. Because of the manner in which processes are defined in the system, message passing is one way, ie parent to children, and there is no reverse communication. The other aspect of communication to note at this stage is that it follows a one to many pattern: one parent needs to communicate with a minimum of two offspring at the same time. Fig.5.2 shows this in diagrammatic form for the example given above.

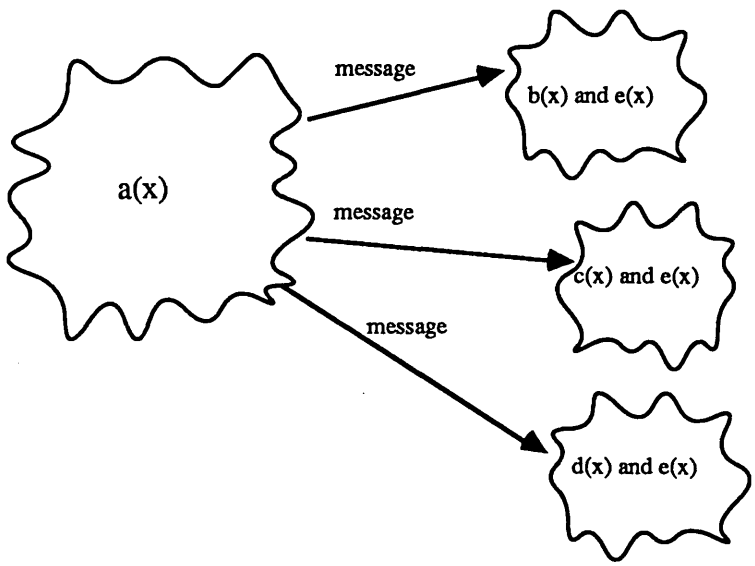


Fig. 5.2 - Process Representation

Because of the requirement that processes are fully independent of each other it follows that the messages which inaugurate them must contain all the necessary information for them to start execution. In Chapter 2 the concept of "environment" for a process has been described: for an OR process in Prolog this consists of the current goal list and binding values, in the PLL an expression tree and binding values. This is the point at which models designed specifically for shared memory machines have a considerable advantage as the transfer of the environment from a parent process to its offspring can be achieved by using shared memory rather than a message containing the necessary information. As the aim in defining intercommunicating parallel execution systems usually favours forcing the computation/communication balance in the direction of computation, the question of representing the environment in a message passing system is of prime importance. In order to avoid large communication overheads it is necessary to condense the environmental information into an optimised message format.

As discussed in Chapter 3.1.3.3 the problem with non shared memory systems is that data on the environment which has to be made common to two processes must either be copied or recomputed. The approach taken in this project is that shared memory machines are too limiting for systems which display large potential for parallel execution, such as OR parallel Datalog programs. Hence the computational overheads of copying and/or recomputing the parental environment have to be accepted but reduced as much as possible.

In the PLL system the environment of a process can be regarded as comprising two parts: the expression tree on the evaluation stack, and the binding values in the variable area of the stack as indicated in the binding list. Because the model is designed for a non shared memory system each process operates within its own independent environment. At any stage during query evaluation these two aspects represent the state of the computation and thus information on them must be passed to offspring processes at the time of spawning. Two points in connection with the expression tree need consideration at this stage: first that the expression tree is not in a suitable format to be passed between processes, and a mechanism for representing it in a linear form must be devised. The linear message will need decoding by the recipient process in order for the expression tree to be re-created. This process is analogous to parsing a query. Secondly a method to keep the part of the message which describes the expression tree as small as possible must be found.

Two methods of cutting down these overheads have been simultaneously employed; one involves the introduction of an optimised message packet which in turn leads to a degree of recomputation. As this is tied to the architectural considerations it will be discussed in Chapter 6.

The second method is to assume that there is a copy of the interpreter plus user defined rules globally available on a read only basis. The most likely implementation of this is to hold a copy of the rewrite interpreter in each processing element. The interpreter consists of the meta or system defined rewrite rules plus the user defined rules which are stored in the rule network as described in Chapter 4.6. At this stage no distinction is made between user defined structures representing base predicates or relations and those which define higher level user "rules", the assumption being that both types of information is immediately available. In a realistically large system it is a reasonable assumption that most of the base predicates would be stored on disk. The memory storage implications for this are discussed in the next chapter. The reason for assuming that each process has an available copy of the interpreter to refer to is that it enables part of the process environment to be described by pointers into the rule network, thus making the process representation more compact. The problem of bindings however still remains. In a model that does not encompass sharing evaluation memory space, binding values have to be included in full in the process creation message. The amount of data that this will involve obviously varies considerably. In systems where heavy reliance is placed on large structured terms it will make for unwieldy communications.

The process based nature of the system is shown diagrammatically in Fig.5.2; this represents processes at the computational model level. The next step in the design is to move to the second level, ie the implementation of the parallel interpreter, and look at the manner in which the abstract concept of independence of processes can be incorporated into the rewrite rule system. This is discussed in the next section. The final level, ie the mapping of the language system onto a parallel architecture and a simulation of its performance, is the subject of chapters 6 and 7.

Two important aspects concerning the model can be seen in Fig.5.2. First the independence of processes means that there is no concept of ordering of process execution. As far as the theoretical system is concerned the processes can be evaluated in any order without effecting the validity of the final outcome. In a theoretical parallel system where processes are evaluated as soon as they are created, the effect is comparable with a breadth first search of the solution tree. In a "real" system computational resources are unlikely to be adequate to provide for simultaneous execution of all available processes, and some form of scheduling will be involved. The independence of processes means that different scheduling schemes can be tried out without any worries about the correctness of the system.

The second feature of the model that the diagram shows is the replicated evaluation of the mutually conjoined expression, ie e(x). This would appear to produce a significant overhead in the amount of computation taking place in the system, although if processes were all being evaluated simultaneously the overall time to produce the query response would not be diminished. This would seem to indicate that the first approach to OR parallelism as described in Chapter 5.3.1 could produce a more efficient system. In fact the amount of repeated or redundant computation is often less than initially expected. Because of the manner in which AND node rewriting takes place, in the situation where b(x), c(x) or d(x) produce FALSE results, no evaluation of e(x) is attempted. If however b(x), c(x) or d(x) are themselves rewritten to other expressions and produce individual and different bindings, these need to be involved in the rewriting of e(x) from the start. Thus in these two situations there is no unnecessary repeated evaluation of e(x). The occasion where redundant computation does exist is when neither b(x), c(x) or d(x) is further reducible: in that instance e(x) will be evaluated three times under the same environmental conditions.

Because of these considerations it has been decided that when the new expression tree is set up in the new process the expression representing the alternatives is placed on the left hand arm of the AND node, thus ensuring that the interpreter will attempt to rewrite it first. In the following two cases the spawned processes will have the same expression trees to work on:
s(x) and (r(x) or t(x)),
and
(r(x) or t(x)) and s(x)
will both result in these two processes
r(x) and s(x),
t(x) and s(x).

Because of the manner in which the conjunction rewrite rule works this will ensure that the alternative subexpressions, ie r(x) and t(x) are evaluated first in the two spawned processes (see Chapter 5.4.4.5).

5.4. The Implementation of the Parallel Process Model

5.4.1. Introduction

The implementation of the parallel process model has involved the design of a modified PLL interpreter. The basic principle of successively reducing a expression until it is in a minimum form by the employment of rewrite rules is maintained, but the system must recognise the nodes representing alternatives, halt rewriting and spawn new processes. The first step however is the move to a process based system.

A process is initiated by a self contained data packet which is received from another process, or in the case of the initial process is constructed from the query. Because the data packet holds all the information required to set up a new process it can be considered as a representation of the process. The first job of the interpreter is to convert the data contained in this packet into an structure that is recognisable to the rewrite rules, ie an expression tree. The interpreter then applies the appropriate rewrite rules to the expression until such a time as it is no longer reducible or it encounters an alternative node. In the former instance it prints out the results, in the latter it halts rewriting and spawns new processes before terminating.

From this outline description it can be seen that the new interpreter has to perform functions that were not present in the original sequential version: first it has to recognise alternative nodes and react to them in a different manner, and secondly it has to handle the construction and decomposition of the data packets representing processes.

There is a third function that the new interpreter system has to perform that is not directly involved with the rule rewriting aspects: it has to provide management for spawned processes. Because the interpreter is actually running on a single processor, the system has to store up spawned processes that are ready for execution and provide some method of scheduling. In the last section it has been shown that the order of evaluating processes is irrelevant to the correct functioning of the logic system, and therefore all that the scheduling algorithm needs to provide at this stage is a method of ensuring that all processes do get evaluated. This aspect of the interpreter's functioning is different from its main operation of performing rule rewrites. As such it is important to maintain a conceptual separation between them. It involves the use of data structures and functions to perform this task of controlling the system, and strictly speaking these should not be regarded as belonging to the interpreter as they would be redundant in the event of the system being used on a "real" multiprocessor architecture. In Chapter 7 it will be seen that it is necessary to add a third layer of simulation in order to model the behaviour of a physical machine.

The data structures and functioning required to accomplish the control of the system, and process spawning with its associated packet formation are described below.

The new parallel interpreter was defined in a separate module which interfaced with the sequential system and eventually with the parallel machine simulation (see Chapter 7.4.1). The parallel rewrite interpreter involved almost 1000 lines of C code and occupied 40Kbytes. Appendix F contains examples of the coding of some of the more important functions used in the parallel interpreter.

5.4.2. Modelling the Parallel Interpreter on a Single Processor

Although the role of the new interpreter is to create and execute independent OR parallel processes, the system is implemented on a single processor system and it is therefore necessary to provide some method of providing this pseudo-parallel execution of processes. Before considering the details of process creation and spawning, it is necessary to look at the modifications that are used to model the running of parallel independent processes on a single processor system. This will be further expanded in Chapter 7 where the parallel machine simulation is discussed.

There are two aspects to the modelling of the interpreter on a single processor: the first has been touched on, namely the storage and scheduling of processes. The second is the allocation of memory space for each process to use while executing. It is assumed that in a real machine each process will be mapped onto its own specified processing element and operate within a private memory in that processing element. However at present the interpreter has to operate in pseudo parallel fashion, ie the system has to model parallel operations on a single processor and memory system.

It has been shown that there is no need for a complicated scheme for scheduling processes at this stage: processes are fully independent of each other and therefore the order in which they are executed does not affect the results of query. As will be seen in the next section processes are represented by five field data structures which have been designed to be held on a linked list. In order to organise process scheduling a global queue of processes awaiting evaluation is defined (the "ready_to_run_queue"), and when processes are spawned they are placed on this linked list. The software that controls the system removes one process from the queue, passes it to the rewrite interpreter which is then responsible for its execution. The system continues in this fashion until there are no more processes in the queue. When the parallel machine simulation is designed it is necessary to include rules to determine the next process to be removed from the queue - these will be looked at in Chapter 7.

The question of allocation of memory space for each process has been the subject of some concern. The computational model defines each process as working in its own environment and this implies that it has available a unique evaluation stack and binding list (the variable list is concerned with user introduced variables and can thus be considered to be globally available). Clearly it is not feasible to divide up the available memory in such a manner as to give each process its own "new" memory space: first because it is not known in advance how many processes will be produced for a given query, and secondly the wastage would soon mean that the system would run out of space. However a system must be designed which allows each process to have its own "virtual" stack and binding list.

This need to allow each process to work in its own environment has been implemented by giving each executing process full control of the general evaluation stack and binding list as defined for the sequential interpreter. When the process terminates, the stack is reset and the binding list cleared, and the next process to execute uses the same space. Theoretically this is a straightforward implementation of the need to model many independent processes in a single system. In practise it has been more difficult to achieve. The reason for this is that as the system has developed the evaluation stack has been used to store certain control information such as the ready_to_run_queue. Whereas the purists would frown at this approach it has ensured that maximum use has been made out of available memory in a situation where there has not been sufficient memory to run as large test programs as desired. However the result is that when a process terminates and conceptually the stack is reset, the reality is that quite careful garbage collection has to be performed rather than a global reset.

5.4.3. Process Representation

The data structure designed to represent a process has to hold two types of information: it needs to incorporate the environmental details (the expression tree and any bound values) in order that process evaluation can be initiated. It also needs to hold a certain amount of control information that is required by the system to organise the scheduling of the process. In Chapter 7 it will be shown how the control information is used in mapping the processes to the physical architecture of the parallel machine. However from the standpoint of the parallel interpreter most of the control information is irrelevant.

The structure representing a process has been defined as a five field block as shown in Fig.5.3. Field 0 holds a unique global process identifier; fields 1 and 2 are discussed in Chapter 7. The "Next" pointer in field 3 allows processes to be queued up as a linked list and held on the ready_to_run_queue. Field 4 holds a pointer to a structure known as the "process description". It is this structure that contains the data required by the interpreter to evaluate the process.

0	1	2	3	4
Proc_no	Control	Information	Next	Proc_desc

Fig. 5.3 - Process Structure

The process description component of the process structure consists of a linked list of bipartite structures holding pointers which represent the conjoined subexpressions of the expression to be rewritten. If any variables are bound the bindings are attached to the end of this list. The process description relies on the fact that the rewrite rules are globally available on a read only basis. This allows a rule to be represented in the process description by a pointer into the rule area of the stack. The assumption is that the rule address will be meaningful to all processes throughout the system and thus the transfer of an address from one process to another is the equivalent of passing the rule name and other data about it. In a similar fashion the initial query is assumed to be globally recognisable. The section on the multiprocessor architecture indicates that this can be achieved without loss of efficiency.

The simplest example of a process description is one which contains only one pointer and no bindings. This type of description results from the situation where the interpreter has encountered a simple OR node in rewriting an expression. For the system that holds the rule
define a(x) tobe b(x) or c(x)?
rewriting of a(x) will produce the expression tree shown in Fig.5.4.

The two resultant processes formed after spawning will have process descriptions as shown in Fig.5.5 where p1 and p2 are the pointers to b(x) and c(x) and where there are no variables bound at the time of spawning.

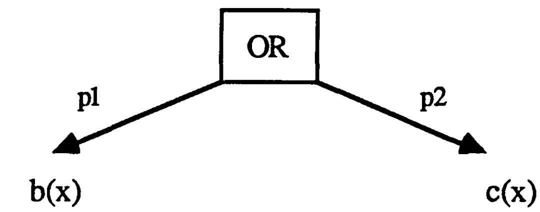


Fig. 5.4 - Expression Tree for b(x) or c(x)

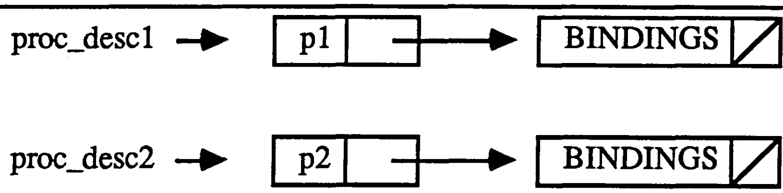


Fig. 5.5 - Process Descriptions

A more complex process description can take the form (Fig.5.6): in this instance the list of pointers represent conjoined subexpressions and the gives rise to the tree shown in Fig.5.7.

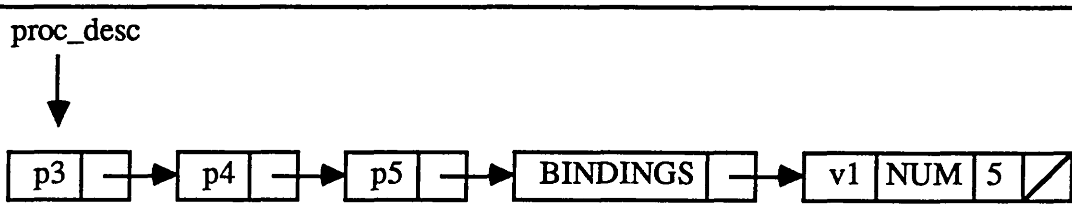


Fig. 5.6 - Process Description

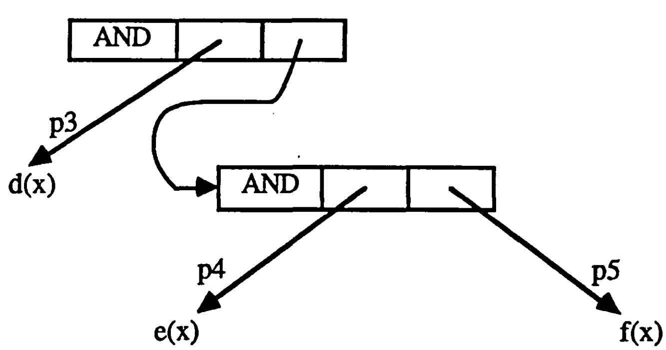


Fig. 5.7 - Expression Tree for d(x) and e(x) and f(x)

The information contained in the four part node after the BINDINGS tag is the data on the variable v1 which is bound to the value 5. The first field gives the stack base offset value, and NUM refers to the data type of the value.

The pointers in the process description are implicitly conjoined and initially it was believed that a pointer to the static rule base or query area would cover all possible logical subexpressions. However the representation of negated expressions has had to be reconsidered. If the rule base contains the definition

define a(x) tobe (not(b(x))) or c(x)?

the pointer in a process description representing the first subexpression, ie (not(b(x))) can give the address of the NOT node in the rules area (see Fig.4.2). However in the case of the rule

define a(x) tobe (not (b(x) and c(x))?

the first rewriting of this expression will result in negation being moved downwards in the expression tree, ie

(not(b(x)) or not(c(x)))

as shown in Fig.5.8. In this case the pointers to the rule for b(x) and c(x) in the rule base have to carry a tag to indicate that negation has taken place.

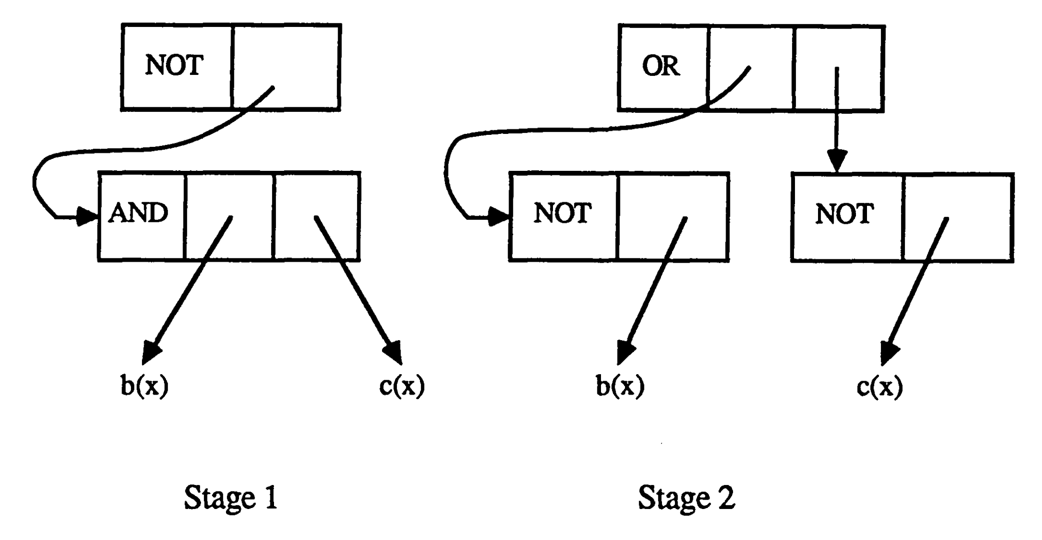


Fig. 5.8 - NOT Node Rewrite

This representation of processes was designed to meet the implementational needs of the abstract parallel interpreter, and in the

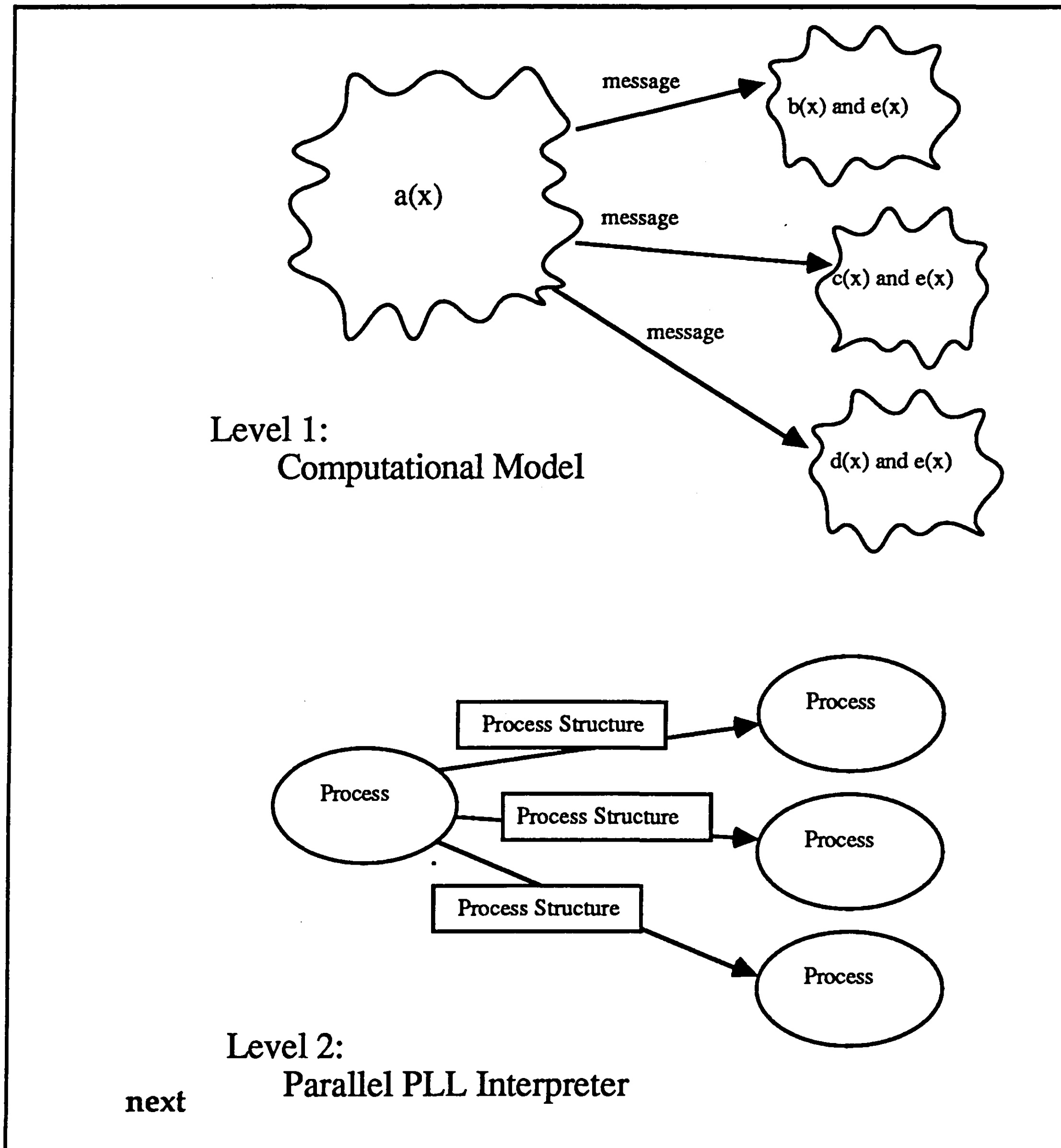


Fig. 5.9 - Process Representation (Second Level)

section the manner in which the process description is constructed and decomposed is looked at. However it is important to note here that this is the second "level" of the implementation of the process model, and as such still represents an abstraction of the system that would be used in a "real" parallel machine. It is nevertheless an executable abstraction and the software to implement this has been produced. These process descriptions are further refined in the third layer of the design in order to model the optimised transfer of information from a parent process to its offspring in the situation where they are genuinely running on separate processing elements. Fig.5.9 shows the process representation design with the second level of implementation included.

5.4.4. Process Spawning

5.4.4.1. Introduction

The creation of a new process takes place either at query insertion time or at process spawning in the event of an alternative branch being encountered in the search space. The typical situation of process creation takes place when spawning occurs and always involves the formation of two or more processes.

The handling of alternatives in the parallel system has to provide the mechanics for process spawning and thus differs from that in the sequential system. This has meant that new rewrite rules for these situations have had to be defined. In practical terms this has led to the introduction of a new "parallel rewrite manager" module to the system which replaces the core interpreter in the parallel system. Some of the inbuilt rewrite rules contained in the original sequential module have had to be completely rewritten, and others modified. (The requirements of the architecture simulation have also meant that alterations to the system rules have been made: this is discussed in Chapter 7). The nodes in which alternative branches can be represented are OR, IN and RANGE. It has also been necessary to construct a new rewrite rule for conjunction handling in order to meet the commonly occurring situation where an OR node is encountered nested within a conjunction.

5.4.4.2. Rewriting of OR Nodes

When the parallel interpreter encounters an OR node it calls the new OR rewrite rule. Instead of the original method of adding the pointers to the two branches to the global or-stack, the rule calls a recursive function to walk down the expression tree from OR node and create as many process descriptions as there are nested OR nodes (Fig.5.10)

As these process descriptions are produced they are each inserted in the final field of a newly created process structure (see Chapter 5.4.3). The process is given a unique number and at a later stage control information will be added to its second and third fields. As the processes with their embryo process descriptions are created they are stored on a temporary queue.

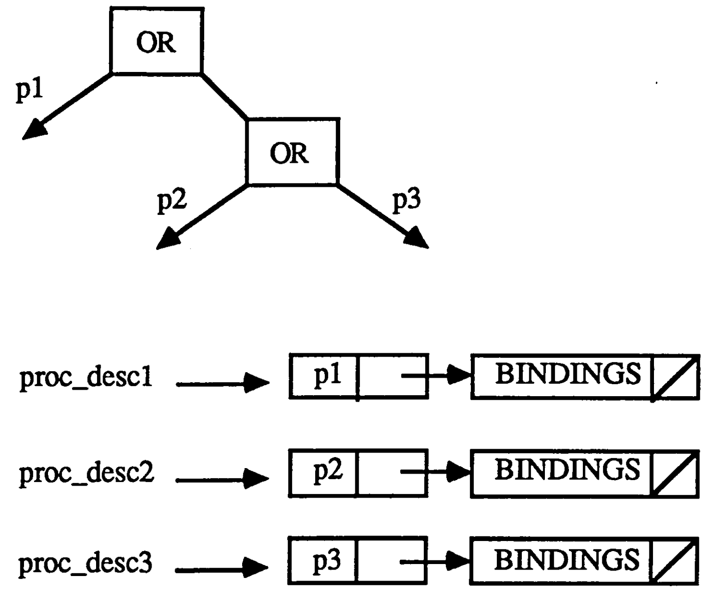


Fig. 5.10 - Process Description Formation

The software then checks in the binding list to discover how many variables are bound and adds them to the end of the process descriptions. At this stage it is recognised that not every binding is necessarily relevant to all process descriptions. In the case where the OR node

a(x) or b(y)

and x is bound to a value, the binding for x is only necessary for one of the resultant process descriptions. However selection of appropriate binding values has not been implemented at this stage, because the final format for the data packet as produced for the multiprocessor machine needs to include all values. This will be discussed in detail in Chapter 7. Thus all bound variables are represented on each of the process descriptions formed.

The newly formed processes are now complete and the final operation involved in spawning is to transfer them to the ready_to_run_queue where they await scheduling for execution. The temporary queue is reset and the parent process now terminates.

5.4.4.3. Rewriting of IN Nodes

The new rewrite rule for IN node evaluation works differently from the original sequential version where the node is successively transformed into a disjunction of equality and a modified IN node, ie

x in [1 2 3]?

is rewritten to
(x=1) or (x in [2 3])?
Because of the need to define alternatives for parallel execution, the new IN rewrite rule works by immediately transforming the IN node into an OR expression tree of equalities (see Fig.5.11).

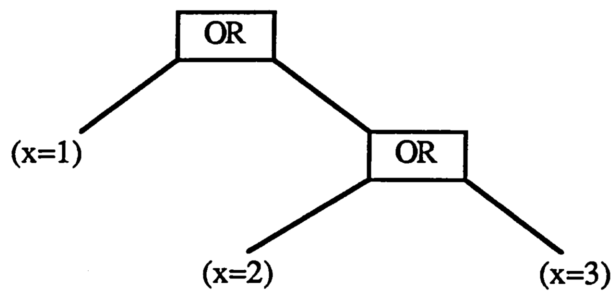


Fig. 5.11 - IN Node Transformation

This expression tree is then passed to the new OR rewrite rule which spawns three independent processes as described in the previous section.

In this way the principle of membership rewriting is maintained but the full expansion into equality relationships is done in one step.

5.4.4.4. Rewriting of RANGE Nodes

The RANGE node is used in association with the IN node and represents the range of integer values that a variable can take; it is denoted by the symbol "..". Thus the query
x in [3..5]?
is answered by
(x=3) or (x=4) or (x=5).
The query
2 in [1..x]?
will produce the result
(x=2) or (x > 2)?

It can be seen from these examples that the final result of rewriting a RANGE node is a disjunction. The sequential interpreter performs the rewriting in a number of different steps in a similar fashion to IN node

evaluation. The new rewrite rule bypasses this serialisation and produces the OR tree immediately on encountering a RANGE node. Thus
x in [1 3..5 7]?
is transformed into
(x=1) or ((x=3) or (x=4) or (x=5)) or (x=7)
and this is passed to the disjunction rewrite rule which spawns the new processes in the normal fashion.

5.4.4.5. Rewriting of Conjunctions

Because of the need to spawn processes whenever an OR node occurs nested within an AND node a new conjunction rewriting rule has had to be defined. In this situation the spawning of processes must involve two operations: first the setting up of the process structures with process descriptions corresponding to the alternative branches or OR nodes. This has to be followed by the task of walking back out of the conjunction "collecting" the conjoined subexpressions that were present at the time of process spawning. These are then added to the end of each process description.

In order to understand the method used to achieve this it is worth considering the spawning of processes in the following examples.

The operation of OR process evaluation as described above works correctly for "top level" OR nodes such as
a(x) or b(x) or c(x) or (x=1)?
However when an OR node is encountered within an AND node (see Fig.5.12) eg
a(x) and (b(x) or c(x) or (x=2))?
the manner in which the original evaluation of AND nodes is performed, ie try-lhs-rewrite, try-rhs-rewrite until no more bindings made, would produce
Step 1: rewrite a(x) - not possible, not in rule base
Step 2: rewrite OR node - leading to three processes with process_desc as shown in Fig.5.13.

In other words these process descriptions give no indication of the conjoined expression on the left hand side which is a proper part of the newly spawned alternative processes. A method of producing the correct

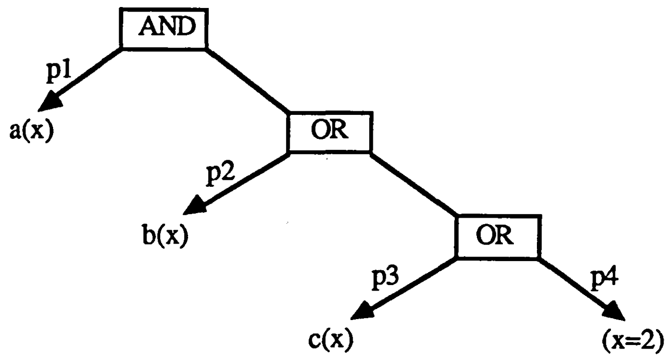


Fig. 5.12 - Expression Tree with AND and OR Nodes

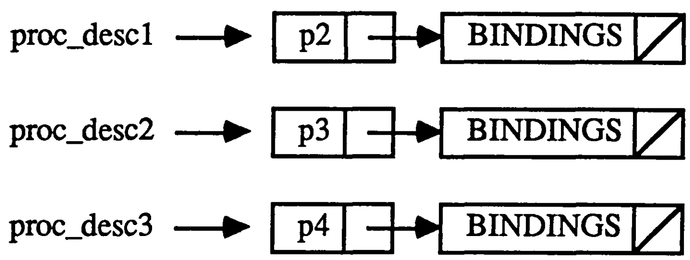


Fig. 5.13 - Initial Process Descriptions

process descriptions (Fig.5.14) is needed when OR nodes are encountered within AND expressions.

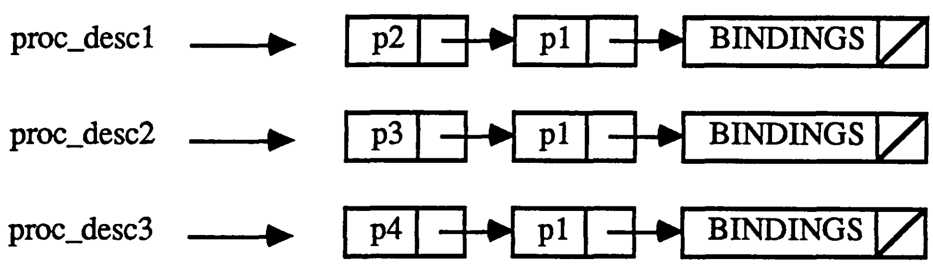


Fig. 5.14 - Completed Process Descriptions

This has been achieved by changing the AND rewriting rule. The new version maintains the rewrite-left, rewrite-right approach until no further alterations are made, but before entering the loop it tests for OR and IN nodes on both its child branches. If an OR or IN node is encountered, instead of entering the loop it spawns new processes in the manner described above, leaving them on the temporary queue, and terminates returning the newly defined AND_OR node. This node type has three fields: its type, plus pointers to both arms of the subexpression tree which has to be included as the mutual part of the process description.

The interpreter recognises this node as signifying that there are spawned processes waiting on the temporary queue which need an extra pointer (or pointers) added to them, and it performs this addition by using the information in the AND_OR node. In the above example the evaluation of the expression
a(x) and (b(x) or c(x) or (x=2))?
would spawn the alternatives as shown in Fig.5.13, and return the node (AND_OR, p1, NULL). This node triggers the addition of the mutual pointer p1 to the end of each process description on the temporary queue (Fig.5.14).

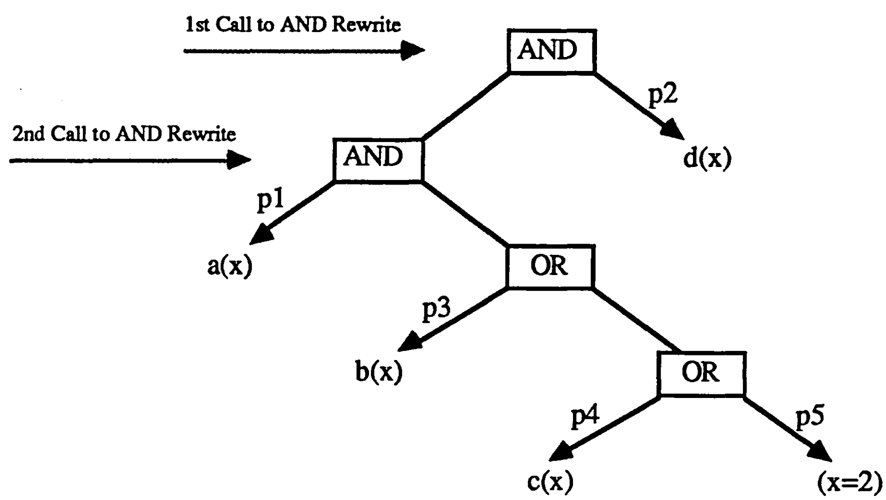


Fig. 5.15 - Rewriting of Expression Tree with AND and OR Nodes

Because of the recursive nature of the AND rewriting rule the AND_OR node returned from a halted AND rewrite may represent a subexpression tree to be converted into a partial process description as can be seen in the example in Fig.5.15. In this case:
2nd call to AND rewrite rule returns: pter1 = (AND_OR, p1, NULL)
1st call to AND rewrite rule returns: pter2 = (AND_OR, pter1, p2).
The result of the top level call, ie (AND_OR, pter1,p2) is then converted into a partial process description (Fig.5.16); this is then combined with the spawned OR processes representing b(x) or c(x) or (x=2) to give the process

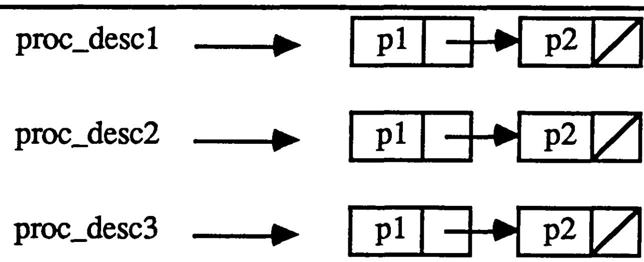


Fig. 5.16 - Partial Process Descriptions for a(x) and (b(x) or c(x) or (x=2)) and d(x)

descriptions as shown in Fig.5.17. Only now are bindings added to the process descriptions giving these final versions in Fig.5.18.

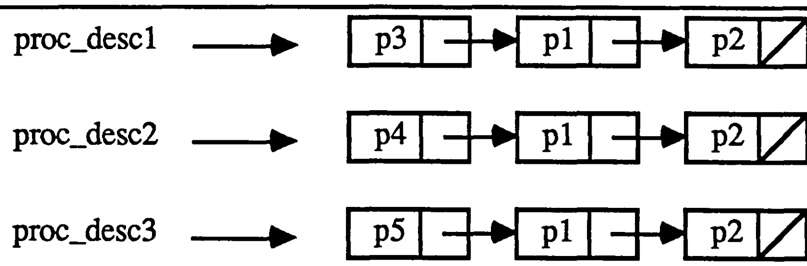


Fig. 5.17 - Extended Process Descriptions for a(x) and (b(x) or c(x) or (x=2)) and d(x)

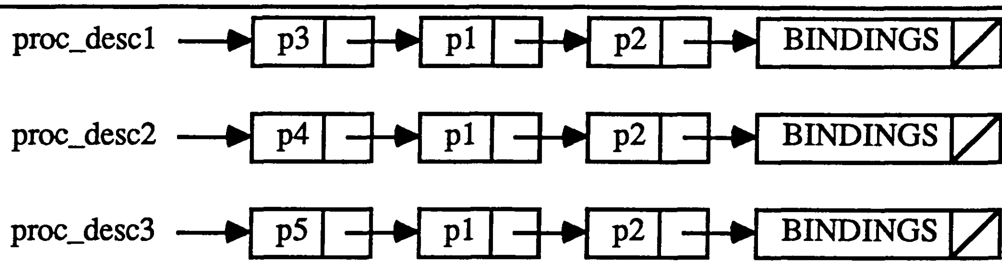


Fig. 5.18 - Completed Process Descriptions for a(x) and (b(x) or c(x) or (x=2)) and d(x)

The general concept of halting execution of an AND expression whenever an OR node is found, spawning processes, and walking directly out of all the recursive calls to collect the other "mutual" branches that have to be conjoined seems to be the appropriate method for creating a linear representation of a tree structure.

There is an additional implementational overhead to this method which is not required in the final design for the multiprocessor architecture.

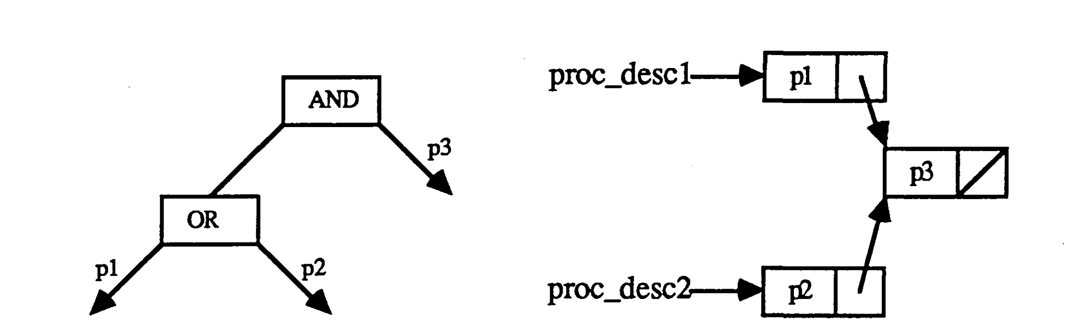


Fig. 5.19 - Initial Process Description Implementation

It is nevertheless important to be aware of it as it gives rise to overheads that have to be discounted in the results on processing times (see Chapter 8). Because the first part of the process description is held in a linked list of specially created two field nodes the situation arises in which the mutually conjoined subexpressions are referred to by the same address when they are added to the process descriptions, eg the AND tree shown in Fig.5.19 produces these corresponding process descriptions.

If the process defined by proc_desc1 is first to execute it will incorporate p1 and p3 into its execution tree and at the end of its evaluation it is necessary to organise garbage collection including the process structure and its associated process description. However the node holding the mutually conjoined part of the process description is needed by the second process which may be evaluated at any stage in the future. In order to simplify the garbage collection and to avoid corruption of information, copying of the mutually conjoined nodes is performed prior to binding insertion, thus providing two fully independent lists as shown in Fig.5.20.

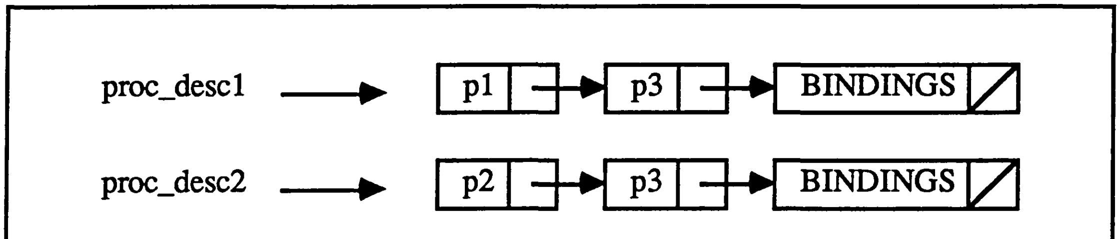


Fig. 5.20 - Final Process Description Implementation

5.4.5. Process Reconstruction

The previous section has shown how a process can be represented in a structure which contains a linear list of pointers and binding values, ie the process description. This has been designed to provide the vehicle for passing information from one process to another. The procedure for setting up a process description has been presented and the situations in which processes are spawned have been detailed.

The other operation in relation to a process description is the complementary one of converting it into a format that is recognisable to the rewriting functions of the interpreter. Once a process has been scheduled for execution and removed from the ready_to_run_queue, the first function of the new interpreter is to convert the process description into an expression tree and to reinstate any bound variable values. This operation is equivalent to parsing an incoming query but involves far less computational effort as the values in the process description represent pointers to nodes that already exist in the system. The only nodes that have to be created by the interpreter are the new AND nodes for the conjoined pointers. Thus the process description shown in Fig.5.21 involves the formation of the tree in Fig.5.22.

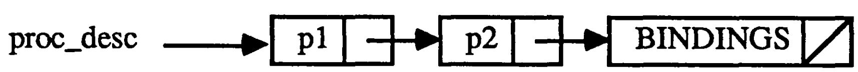


Fig. 5.21 - Process Description

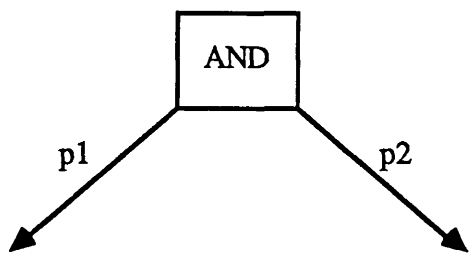


Fig. 5.22 - AND Expression Tree

The second operation that has to be performed before process evaluation can take place is the copying of binding values into their position on the stack and the insertion of the stack addresses into the binding list. The following bindings representation (Fig.5.23) involves putting the integer tag NUM and the value 17 at addresses v1 and (v1+1) on the stack and similarly installing NUM and 44. The addresses v1 and v2 are inserted into the first two elements of the binding list and the value of binding level becomes 2. Having established the environment for process evaluation to start control is passed to the rule rewriting part of the interpreter.

Fig. 5.23 - Bindings Representation

A process can be regarded as having three parts: the setting up of the expression tree, the application of the rewrite rules to the expression tree, and the spawning of processes in the event of an alternative being encountered. The proportion of time taken by each of these operations is the subject of discussion in Chapter 8, but at this stage it is of importance to note that the first operation, ie the conversion of the linear process description into the correct environment, occupies a small fraction of the total execution time.

5.5. Summary

This chapter has described the decision to investigate an OR parallel system for the PLL. The abstract computational model has been defined and a parallel interpreter produced. The new interpreter is based on the sequential version using the technique of applying stored rewrite rules as an inferencing method. However the parallel system is based on the concept of evaluation of independent OR processes. The manner in which processes are represented is discussed and the modelling of parallel execution on a single processor system is described.

Chapter Six

The Parallel Architecture

6.1. Introduction

The previous chapter has described the identification of sources of potential parallelism within the PLL and has shown how one of these (OR parallelism) can be encapsulated in a computational model and its interpreter. In this chapter the next step is discussed. In order to reap any benefit from the change to a parallel process model the parallelism expressed in the interpreter must be mapped onto a suitable multiprocessor architecture. It is recognised that the constraints imposed by architectural considerations are likely to impose limitations on the amount of actual parallel execution that can take place. However the intention is to design an architecture which will support the computational model as closely as possible in an attempt to derive as much benefit as possible from the potential parallelism. The aim of the chapter is to define the functional requirements for a parallel machine based on the knowledge of the operation of the new parallel interpreter, and to present a possible hardware realisation of this design.

The chapter shows how the interpreter can be mapped onto the architecture: in order to test the validity of this design a working simulation of the system has been developed. The simulation is the subject of Chapter 7, and represents an important step in development of the actual hardware system. Resources have not been available during the course of the present project to consider the construction of a prototype machine. The role that the simulation plays in the overall development of the system is discussed in Chapter 7.

This project developed out of work done on the design of a multiprocessor architecture for knowledge bases using semantic networks [Hird 85]. The original intention was that the parallel logic language would be mapped into the type of architecture that had emerged from the earlier work. However an analysis of the patterns of communication involved in the parallel PLL has shown that the needs of the two systems are different. The type of architecture that was believed to be suitable for semantic networks was a fixed topology with nearest neighbour connections. It is not within the scope of this project to comment on its applicability for other types of computational model, but what has become clear is that it is not suitable for the implementation of a parallel PLL system.

The chapter discusses the reasons why a fixed topology architecture is not suitable for the parallel PLL. Out of this analysis has come a clearer understanding of the functional requirements of a suitable multiprocessor system. These are discussed and the mapping of the parallel interpreter onto the functional design is described. Finally the chapter looks at a hardware realisation of the functional design.

6.2. Fixed Topology Architectures

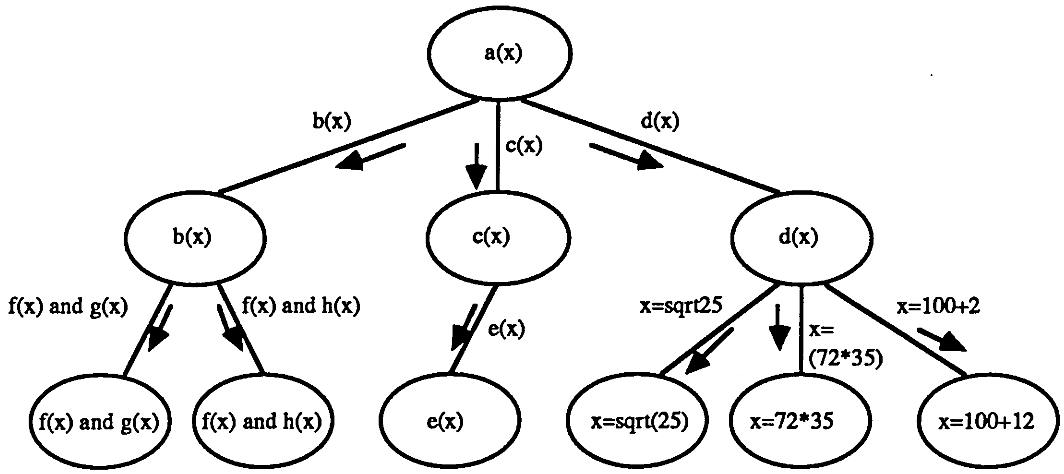
The original design for the multiprocessor architecture specified a two dimensional rectangular array of processing elements with nearest neighbour connections and this was intended to form the basis of the design for this project [Loh 82]. It was hoped that this would prove a suitable design for a parallel PLL system, as it had considerable advantages from the hardware implementation point of view. However attempts to map the computational model for OR parallelism to these hardware proposals gave rise to a number of problems. Chapter 3 has looked at various architectural proposals which have been put forward for parallel logic language systems. Having decided to concentrate on non shared memory systems because of scalability problems with shared memory machines it is worth looking again in more detail at the suitability of fixed topology distributed memory architectures.

The term fixed topology architectures is used here to mean the type of non shared memory machine in which the connections between the different processing elements are not dynamically reconfigurable at run time. This definition clearly applies to nearest neighbour grids machines, but it also includes systems such as the Parsifal architecture [Capon 86], [Hughes 86]. Although in the Parsifal system the individual processing elements are connected by means of a bank of crossbar switches which enables any Transputer to be linked to any other, this reconfiguration is generally done prior to run time and remains fixed during the course of program execution. The hypercube architectures such as the Connection Machine and the iPSC, offer a fixed pattern of inter processing element connections, but the "richness" of the communication network means that messages can be sent between distant nodes using shorter paths than in a nearest neighbour grid [Hillis 85], [Intel 86]. For all fixed topology grids whether nearest neighbour or hypercube, methods of efficient message routing need to be employed.

A fixed topology architecture has advantages for the hardware designer and manufacturer, and can produce good performance benefits for the appropriate application. The identification of suitable applications then becomes the subject of research: the use of the Connection machine for text retrieval work is an example of a study to redefine an application to match the processing capabilities of a parallel machine [Stanfill 86].

The reasons for believing that a fixed network of processing elements is not the most suitable architecture for the parallel PLL lies in the pattern of communications involved in the language. A fundamental aim in defining a parallel system is to maintain a high processing to communication ratio, and to ensure that processing is held up as little as possible by delays in the receipt of data from other processing elements. It is generally true that in this type of architecture communications between directly connected processing elements are more speedy than those which have to be routed through a number of other processing element nodes, and it is this aspect that leads to one source of inefficiency when looking at the execution of parallel logic languages on this type of machine.

Fig.6.1 shows a simple solution tree to a query put to the parallel PLL. The nodes represent processes and communication, ie message passing, takes place along the arcs. In order to execute this efficiently, the ideal mapping would be to place each process on a separate processing element and map the arcs onto the direct physical links between them. However this is clearly not possible, as the form of each solution tree varies from query to query, so a fixed topology suitable for one query would not provide the necessary links for others. Of course even if this exact mapping of the topology to suit the query was possible, it would still be highly wasteful of resources, in that as the tree expands downwards, the higher level processes die and leave their processing elements inactive. The idea of using a system such as the Parsifal architecture which could be customised for each query before run time also looks doubtful, because the nature of the rewriting process means that it is impossible to predict the shape of the solution tree in advance - the so called "non determinacy" of logic programs.



define a(x) tobe b(x) or c(x) or d(x)?
define b(x) tobe f(x) and (g(x) or h(x))?
define c(x) tobe e(x)?
define d(x) tobe (x=sqrt25) or (x=72*35) or (x=100+12)?

Fig. 6.1 - PLL Query Solution Tree

The fixed topology approach has the disadvantage that some communications have to be routed through a considerable number of stages to distant processing elements, because it is not possible to make a sufficiently good mapping to allow processes always to send messages to close processing elements. Because of the unpredictable nature of the pattern of processes, the situation is likely to occur at some stage that the communication delay in setting up a process in a distant but idle processing element is greater than the time to queue it up and execute it locally in a serial manner. Various projects have proposed schemes to minimise this transfer of data across many stages in the network: these involve hierarchical partitioning of the architecture and obviously provide considerable benefit to the efficiency of the system. This type of approach which encourages locality of communication is seen in the Data Diffusion Machine proposals [Haridi 89] (see Chapter 3.2.4.3).

The other problem with communications in the type of fixed network architecture that was originally considered for the parallel PLL is the serialisation of communication. When the solution tree diagram for a PLL query is looked at it becomes clear that when a parent process spawns offspring processes it is ready to communicate with a number of processes simultaneously. However the type of architecture that requires messages to be despatched from one node on a number of different routes or connections will inevitably serialise the operation. If the number of offspring processes is large this serialisation may account for a considerable amount of processing time within the spawning process. In this situation communications times within the system will depend not only on the length of each data packet but also on the number of processes spawned (offspring) rather than the number of spawning (parent) processes.

6.3. Functional Requirements of the Multiprocessor Architecture

The pattern of communications produced by a query to the parallel interpreter was the crucial aspect in the decision to reject a fixed network hardware design as being inappropriate for the ideal parallel PLL system. The communications within the parallel system display the following characteristics: they are

a) unidirectional,

b) one to many,

c) unpredictable at query insertion time.

The first two characteristics indicate that an architecture with broadcasting capability would be appropriate, and the third feature means that the communications links between processing elements should be dynamically reconfigurable during query execution. These requirements are discussed in [Brown 89] and form the basis for the work on the hardware design.

The broadcast mechanism within the machine must be capable of providing the one to many communication pattern for process spawning and in addition should support multiple broadcasts as there are potentially many simultaneous spawning operations. The broadcasting of information from one processing element to many others allows the spawning of multiple processes to take place in one operation on the assumption that a format for the message or data packet can be designed that conveys the appropriate information for all the processes. Thus the serialisation of data packet transmission can be eliminated. The format of the packet is discussed in detail in Chapter 6.4.3 but it is important to note here that much of the information needed to initiate each individual process is common to all, because it represents in part the parental environment at the time of spawning.

By designing the system to meet the communication needs as closely as possible, the overheads involved in data transmission should be kept to a minimum. However this is not the only factor that influences the performance of the machine. The other crucial aspect is the load balancing between the different processing elements. (This ignores for the present that the actual rewrite code may contain inefficiencies - see Chapters 8 and 9). It is assumed that in any real machine the processing resources available are not going to be sufficient to ensure immediate task execution at all times. In terms of the parallel PLL this means that during query evaluation, probably for the majority of the time, there will be more processes ready for evaluation than there are idle processing elements. Thus the architectural design has to incorporate facilities for load balancing or scheduling of processes. It has been seen in Chapter 5 that this task is made more difficult by the fact that processes vary considerably in the number of computational steps they take and this is not predictable in advance.

Having identified these requirements for an ideal architecture the question of the implementation of the design concepts can be regarded as having two stages. The specification of the architecture at a functional level is the first step. At this point the implications for mapping the computational model onto this system are looked at and the interpreter modified where necessary. If this mapping process is achieved successfully the final design phase represents the detailed hardware specification. In practise the separation of the two stages is not clearly defined: it is a pointless exercise defining an architecture at the functional level if it is not technically feasible to implement.

6.4. Functional Design of the Multiprocessor Architecture

6.4.1. Introduction

Fig.6.2 shows the main functional components in the proposed multiprocessor architecture. Query evaluation takes place exclusively in the processing elements. It is anticipated that the system will contain a substantial number of processing elements, possibly upward of a hundred. Each processing element node holds a copy of the rewrite interpreter including user defined rules. As discussed in Chapter 5.4.3 the initial query is also assumed to be common to all processing elements, this having been achieved by a global broadcast following the initial parsing operation.

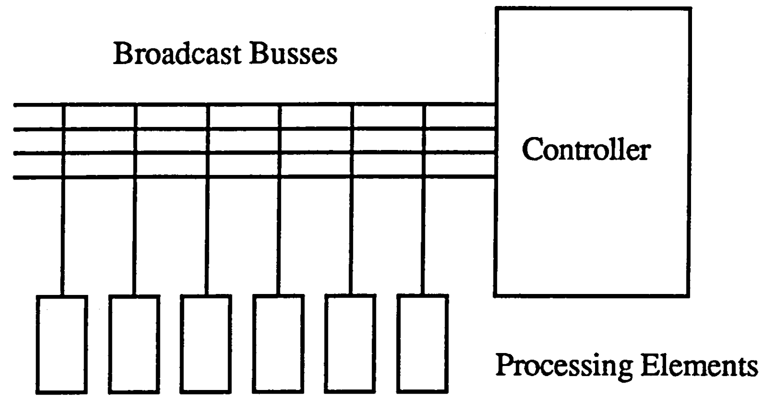


Fig. 6.2 - Functional Outline of Multiprocessor Machine

The broadcast capability is implemented by a multiple bus system which can be configured to allow any processing element to broadcast to all the others or to any designated subset of them. The inclusion of multiple busses allows for several such broadcasts to be performed simultaneously. The number of busses required to give optimum performance is discussed in Chapter 8 in light of the simulation results.

The controller unit has two main functions: the configuring of busses to allow the appropriate broadcast to take place, and the allocation of processes to processing elements based on a measure of the work load in each processing element, ie load balancing. It also acts as the interface between the user and the parallel machine.

6.4.2. Query Evaluation

The operation of query evaluation involves the following steps: the query is set up as a process in a designated processing element, where the rewrite interpreter proceeds to evaluate the expression as described in Chapters 4 and 5 until process spawning occurs. The basic method of process spawning remains as described in Chapter 5 but has been modified slightly to meet the needs of the machine. Instead of producing n process structures representing n processes, the interpreter constructs a single data packet which incorporates all the data representing the n processes. The parent process then terminates and a request is made to the controller for a bus to make a broadcast of the data packet. In the more straightforward of cases this will involve broadcasting to n processing elements. The n processing elements are alerted that they are to receive a data packet and the parent process is given control of the bus. The data packet is then broadcast simultaneously to all n processing elements and the bus is de-allocated. The receiving processing elements store the data packet on their internal queue of processes awaiting execution and in due course it will be scheduled for execution. The interpreter has to be modified to handle the combined data packet, which involves distinguishing which of the n processes it is responsible for. The method used is described in the following section.

6.4.3. Data Packet Definition

In order to ensure that the communication system is not swamped with lengthy data packet transmissions, it is important to consider the optimal form of the data packet. The packet needs to hold all the information required to inaugurate new processes but at the same time it must be as small as possible. The manner in which process structures are defined in the abstract interpreter is clearly not suitable for the real multiprocessor machine: the availability of a broadcast mechanism means that the process spawning information can be passed to many processing elements simultaneously if a method can be found to incorporate the data for all the offspring processes into one message. This section looks at the definition of an optimised data packet.

Several initial assumptions can be made about the pattern of communications and the availability of local data:

a) a processing element can broadcast directly to a number of other processing elements,

b) the receiving processing element can be given some form of advance information concerning the part of the data packet that is relevant to it (see Chapter 6.5.5),

c) the rule base and original query are available to all processing elements.

The new data packet has to contain the information at present held in the group of process structures defined at process spawning time. This information is in three parts:

a) the OR branches in the expression tree which gives rises to the new processes,

b) any mutually conjoined expressions,
c) all bound variable values.
It can be seen from this that b) and c) are required data for the entire group of spawned processes. It is only the information in a) that distinguishes one process from another. The question of data packet design therefore can be divided into two parts: first the optimal manner of representing all the common data, and secondly the method of including information in the combined data packet that will allow individual processes to be distinguished.

The question of condensed representation of the mutual information is considered first. The data packet needs to contain details of mutually conjoined expressions and any bindings. If the following simple expression is considered,
a(x) and b(x y) and c(y) (with x instantiated to 10),
it can be seen that three types of data has to be represented: the variables, the binding values and reference to the rules or predicates involved. In the present interpreter bindings are passed by reference to their general stack location (with their value) as are uninstantiated variables. Although it would be possible to pass data about variables in this format it would mean that all processing elements would need to have the same (long) stack available for variables and also the data packet would be unnecessarily lengthy because of the long values needed to represent the stack address. Hence the decision to represent variables and bindings with a value that relates to their position in the data packet has been explored. The above expression, ie
a(x) and b(x y) and c(y) (with x instantiated to 10)
would be represented by the data packet as shown in Fig.6.3.

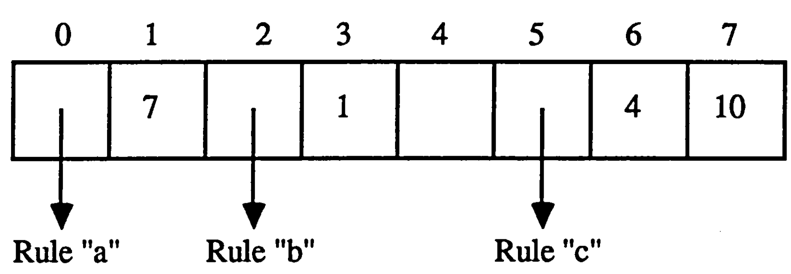


Fig. 6.3 - Data Packet Representation

In the packet the slots marked 0, 2 and 5 hold pointers to the respective rules in the rule area; slots 1, 3, 4 and 6 represent the variables. Slot 1 is the first reference to the variable "x" and slot 4 is the first for "y". The number "1" in slot 3 indicates that it refers to the same variable as defined in slot 1, and similarly the value "4" in slot 6 links the two references to "y". The fact that "x" is bound is shown by the reference in slot 1 to the slot 7. Slot 7 holds the binding value.

When it comes to the passing of information about the rule to be evaluated, the present system represents this by pointers to the nodes in the rule base. This method can be used in the real machine as the rule information is assumed to be available to each processing element. The data packet therefore includes the address in the rules area that allows a receiving processing element to identify the rule to be used. The architecture is thus providing a form of global addressing for the user defined rules.

The method of representing the alternative processes uses the same concept. However instead of including pointers to all the separate branches of a spawned OR tree it would be more efficient to send a pointer to the parent node, and let each processing element identify the child of the OR node that is destined for it from the information received from the controller. This information is obtained from the controller prior to packet broadcast, when the receiving processing elements are alerted that a broadcast is about to be made. This part of the communications pattern is necessarily serial but involves the passage of a very small amount of data. Timing predictions indicate that the operation should not produce an unacceptable overhead for the communication/processing ratio. This is discussed in Chapter 6.5.5, and in Chapters 8 and 9 where results from execution runs using benchmark tests are presented. As far as the evaluation of processes is concerned this method involves a small processing overhead as each receiving processing element (having been informed which OR branch it has to deal with) must follow the pointers from the parent node down to it.

If this latter system is used for OR node representation (ie the sending of the parent node pointer) any variable in the parent node has to be represented in the packet. This representation can be made in the same manner as that used for variables in the mutually conjoined expressions. If one of the child OR nodes has new quantified variables, these will be installed by the receiving processing element at the start of rewriting.

Thus the combined data packet representing the three processes spawned from the following expression:
(a(x) and b(x y) and c(y) and (r(x) or s(x) or t(x)))
(with x bound to the value 10)
is shown in Fig.6.4.

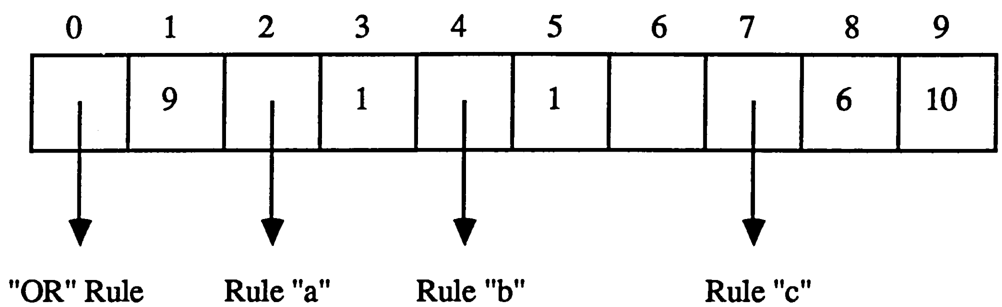


Fig. 6.4 - Combined Data Packet

It is recognised that in a non shared memory machine that data needed in two separate processing elements must either be copied from one to the other or recomputed in the second. The method of compaction of data proposed for the data packet can be viewed as an intermediate between copying and recomputation. The representation of alternative branches of the expression tree by one pointer means that the software which sets up the new process has to "recompute" the branch required. This is not such a major computational task as the recomputation involved in the Delphi approach where each branch of the tree is given a label and the exact position in the solution tree is recomputed each time as discussed in Chapter 3.1.3.3 [Alshawi 88]. On the other hand the inclusion of the details on bindings and the mutually conjoined subexpressions means that these can be immediately installed in the new process, ie this information has been obtained by copying.

6.4.4. Size of Data Packet

It is necessary to look more closely at the design of the data packet in order to establish how much space is required for each element it holds.

The first decision to consider is the identification of the various types of data contained in the packet so that the receiving processing element can decode the packet correctly. This can be done either by tagging each data item with its type or by constructing a header to each packet which defines its precise composition. Both method have been looked at and with the present types of queries there appears to be no advantage in defining a headers. Therefore the simpler method of tagging each data item has been proposed, and this method has been incorporated into the simulation software. It may prove necessary to review this decision at a future stage if realistically large applications are involved.

In order to look at the overheads of tagging it is necessary to see how many types of data items need to be represented in the packet. The obvious items are:

a) pointers into the rule or query area,

b) variables,

c) binding values.

The binding values fall into four categories, ie integer, floating point, list or string values. This brings the total data types to six; however it has been found necessary to use two other tags for user defined variables and negation.

The present system does not need to note specifically which variables are user introduced, ie in the query, and which are the result of quantified variables being introduced during evaluation. This is because the general stack reference is used as a means of identifying variables and therefore the links are maintained between the variable list information and the variables in the process structure. However the move away from the use of the general stack reference means that a tag must be included within the data packet to indicate whether a variable is a user one or not.

The only other information that has to be included in the data packet is that of introduced negation. If a negated expression is in the rule base the pointer to the NOT node in the rule base will pass the information onto the receiving processing elements, but there may be circumstances due to the pushing down the expression tree of NOT nodes in de Morgans rewrites, that it is essential explicitly to indicate negation that is not present in the rule or query area. This has been discussed in Chapter 4.5.5.

The final count of data type tags required is therefore eight which means that the tag requires a three bit space allocation.

The space requirement for the pointer to the rules is dependent on the size of the rule base. Fig.6.5 shows a small table of data obtained for the Sun 3/60 Workstation on memory usage with a variable number of rules. These rules included some base predicates, but these were not of the size expected in a large system holding considerable tables of relations. The whole position of large base predicates requires separate consideration as realistically these are likely to be held in secondary storage not main memory. The data in Fig.6.5 shows that 10 rules will usually occupy less than 2 Kbytes. Based on the assumption that a realistic system might hold 1000 such rules, this would imply memory requirements in the order of 200 Kbytes. However it will be recalled that the manner in which the rules are stored includes the actual string representation of names both for the head of the rule and the body. For large applications based on the PLL it is realistic to expect a symbol table or other intermediate optimisation to be used to reference this information. If the symbol table is implemented the storage requirements for the rules will be reduced considerably. Thus the allocation of 200 Kbytes per 1000 rules is over generous. However taking the present representation a 20 bit pointer will allow 128 Kbytes to be addressed, ie storage for up to 600 rules.

	No.of Rules	No.of Words	Kbytes
Program 1:	10	1059	2.2
Program 2	19	1675	3.4
Program 3	25	1981	4.0

Fig. 6.5 - Rule Storage Data

The question of variable representation is somewhat different: the present system restricts the number of variables a user can include in a query to twenty, and on this basis a 8 bit value (5 bits plus three tag bits) can be used as the data packet value. Variables introduced during rewriting may run to considerably higher numbers. The present benchmark programs are not sufficiently large to enable confident predictions to be made about realistic maxima for introduced variables, and thus an arbitrary value of sixteen bits has been proposed for this, giving the possibility of over 8000 introduced variables in one query evaluation. It seems likely that for most applications this figure would prove unnecessarily generous.

Integer and floating point numbers may be handled as sixteen or thirty two bit values depending on the processor used in the processing nodes. They therefore need a nineteen or thirty five bit space allocation for the data item in the data packet.

String representation is based on the assumption that a symbol table is utilised. Thus one data item (tagged with a string identifier) serves to communicate a string value in the data packet. Realistic estimates of the size of the symbol table are not available because of the nature of the benchmark programs, so the decision was taken to use the sixteen bit data item format allowing for a symbol table of over 8000 entries.

The question of list inclusion in the data packet raised the basic problem that for programs which are heavily dependent on list processing operations copying of variables bound to lists inevitably produces lengthy communications. On the other hand if the intention is to limit the system to the use of strictly defined Datalog programs the problem vanishes as list structures are not permitted. This restriction appears to be too limiting for the system and thus the possibility that list structures may be included in the data packet has to be allowed for. Each list member therefore has to be identified including its appropriate tag. The start of a list has to be marked by a data item tagged as a list and giving the number of members included in the list. On the assumption that the maximum list size is restricted to 1000 items the list enumerator has to be 13 bits space allocation.

The final total of the data types represented in the packet is shown in Fig.6.6.

Data Type	No.of Bits/Data Item (including tag)
Rule/Query Pointer	20
Negated Rule/Query Pointer	20
User Variable	8
Rewrite Introduced Variable	16
Integer	19
Floating Point Number	19
String	16
List Enumerator	13

Fig. 6.6 - Data Packet Type Sizes

6.4.5. Data Packet Implementation

The format of the data packet has been defined and from this the modifications to the process spawning functions originally employed by the rewrite interpreter can be specified. The new process spawning routines have to construct the combined data packet, and in a complementary fashion the process initiation functions have to decode the new data packet in order to reconstruct the appropriate expression tree. However these new functions are not part of the computational model but relate to the functional requirements of the architecture. It has been the intention to maintain the conceptual separation between the software representing the implementation of the parallel interpreter and the software used for the architectural simulation. In view of this the parallel interpreter has not been altered to produce the combined data packet; it still follows the computational model and provides for the spawning of individual processes as represented by process structures.

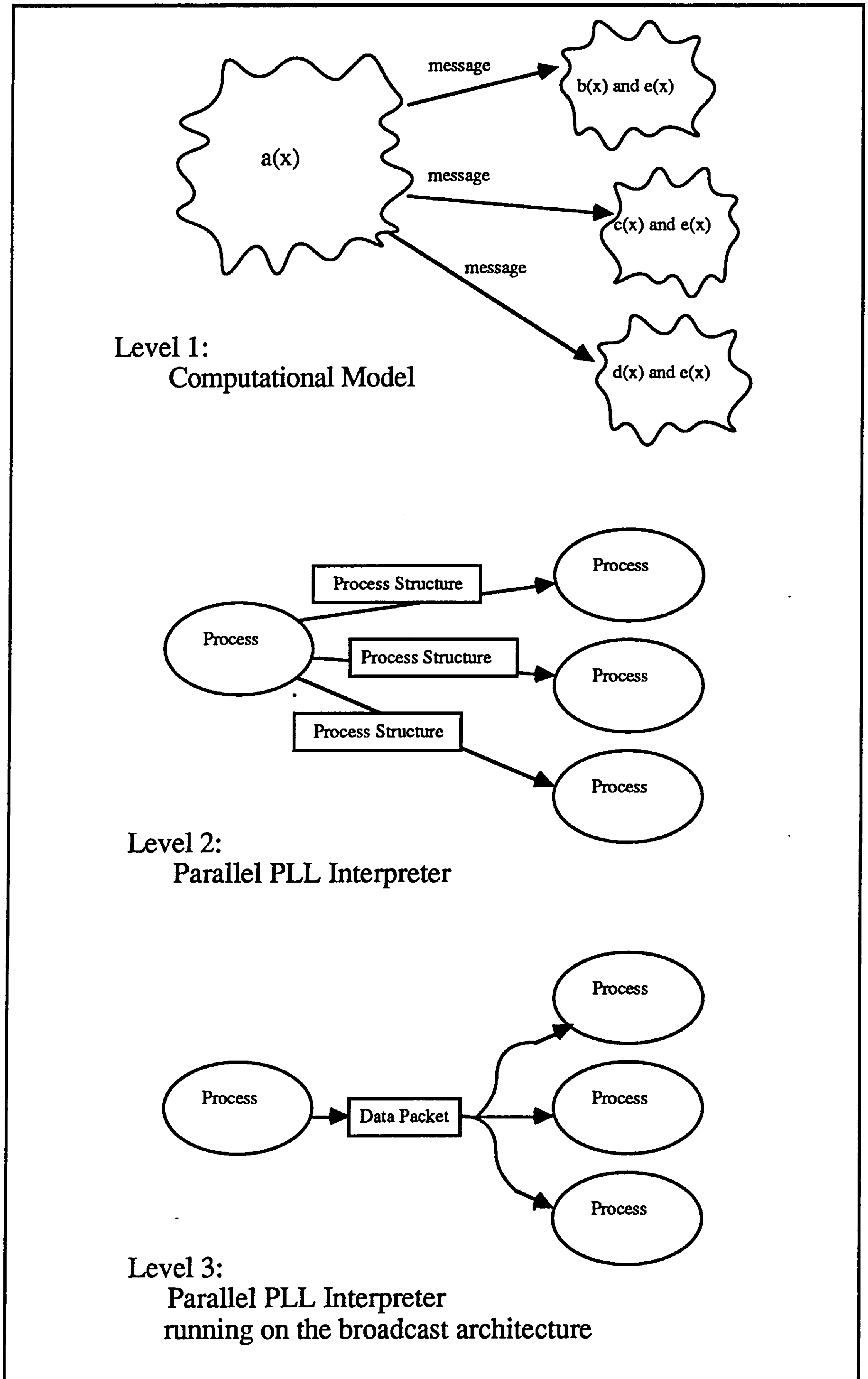


Fig. 6.7 - Process Representation (Third Level)

In Chapter 7 the simulation system is discussed: the system needs information on the size that the combined data packet would be if it were produced. The size of this data packet is needed in order to calculate the length of time the broadcast of a given data packet will take. There is no actual need to construct the packet and therefore functions have been installed which calculate the size of the combined data packet. This information is available from the individually spawned process structures using the data item sizes as defined in the previous section.

The final level can now be added to the process representation diagram (Fig.6.7). This shows the spawning of processes as the data packet representation designed for the parallel multiprocessor architecture.

6.5. A Bus Based Multiprocessor Architecture

6.5.1. Introduction

The functional design of the multiprocessor architecture for the parallel PLL has been a main concern of this project as it has been the intention to produce a simulation of the system at the functional level. The simulation is to provide quantitative results on the predicted behaviour of the machine and should indicate whether the potential speedups in performance over the sequential system make the construction of a prototype worthwhile. For this type of simulation it is not necessary to model the behaviour of the multiprocessor architecture at a low level: essentially data is required on the timing of execution of processes and delays incurred in process execution through contention for the communication network and non optimal load balancing of work in the processing elements. Therefore the work on the multiprocessor architecture has concentrated on its functionality rather than its hardware implementation. However the hardware design has been specified by John Brown and this work is detailed in his report [Brown 89]. The description of the hardware can be considered in two parts: first the aspects that this project has been concerned with, namely the relationship between the controller, the individual processing elements and the multiple bus communication system, and secondly the proposals for accessing data from a multiple disk system. The diagram of the functional units of the machine has therefore been extended to show the disk units (Fig.6.8)

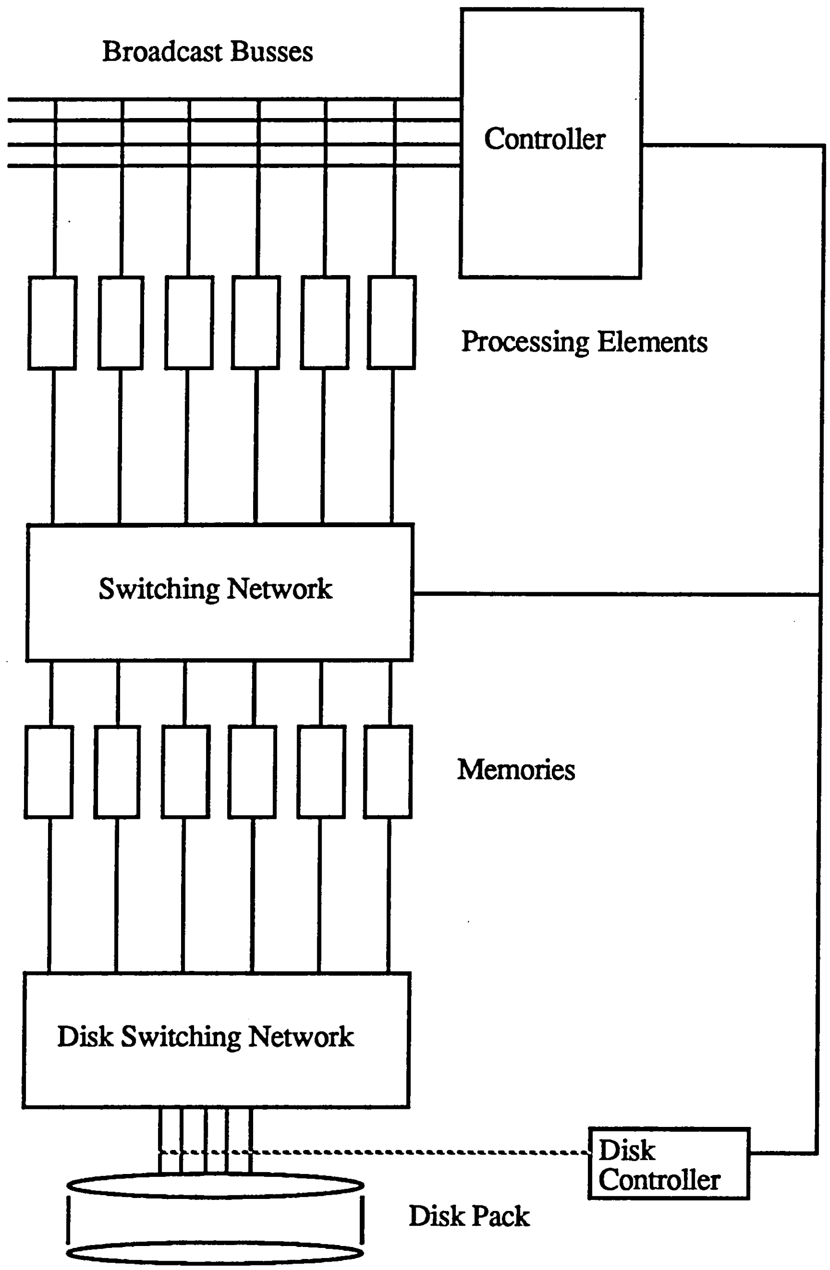


Fig. 6.8 - Functional Design of Extended Multiprocessor Machine

It has been recognised that in a realistic system although the storage of user defined rules in each processing element is a reasonable design feature, base predicates are likely to be held in secondary storage. However as far as the computational model is concerned there is no distinction between an alternative in a high level rule or one in a set of base predicates. Thus the initial system has followed the assumption that alternative versions of rules and base predicates can be found in the memory of a processing element.

In the following sections the design of the basic machine and its relationship to the work done on the parallel PLL are discussed. The storage of base predicates on disk is considered briefly; this is still speculative work and no detailed proposals have been made.

6.5.2. The Multiple Bus Broadcasting System

It has been shown that process spawning can be achieved by the construction of a combined data packet which is then broadcast from its parent processing element to a number of other processing nodes. The concept of broadcasting involves the simultaneous transmission from one processing element into the private memories of the receiving nodes. The operation is made practical by the fact that the serial aspect of the task, ie alerting the receivers, involves only the transmission of a single address whereas the data to be broadcast is considerably larger. The factor that determines the speed at which broadcasting can take place is the reception time in the receiving nodes. Data must be stored into successive memory locations and the settling time of these memories determines the transmission rate. There is no need to broadcast addresses on the bus; this is handled by a local counter in the receiving node. There is no handshaking or feedback path involved and hence the broadcaster can deliver data at a rate to allow for the correct reception, ie for data settling, counter incrementation and address settling in the receiving processing elements.

However the intention is that the architecture should be scalable for large numbers of processing elements and there are likely to be hundreds of potential receivers. Therefore the bus has to be constructed with the use of drivers to support this fanout, and Brown has shown the hardware required to support this tree shaped design. This is similar to the bus system described by Mudge, Hayes and Winnox in which multiple busses are used to access shared memory [Mudge 87]. Because of the tree shaped bus design maximum bus delays increase logarithmically with the number of devices on the bus. However this delay does not affect the maximum rate of data transmission because there is no need for handshaking.

Analysis of the execution patterns in the OR parallel PLL system has indicated that multiple broadcast busses are required as there are likely to be many near simultaneous calls to broadcast data packets. Reference to the ideal number of busses in the system has been deliberately avoided because there is no way of knowing this until some quantitative data is obtained on the timings of process evaluation and predicted data transmission times. One of the main aims in developing the simulation is to obtain this data. It is hoped to obtain a clear idea on the optimum ratio between number of busses and number of processing elements given that hardware implementation and cost may be decisive in imposing limiting values. The use of multiple busses has implications for the design of the processing elements. This is looked at in the next section.

It is worth noting here that this project has taken the approach that broadcasting should be done (if appropriate) after process evaluation has ceased. Broadcasting is used essentially as a mechanism of copying the process environment simultaneously into a number of other processing elements. As such the case can be made for the broadcasting of data during process evaluation. This is the approach taken in the Swedish BC machine project where the memories of designated "slave" processors are updated at the same time as that of the "master" throughout processing [Ali 88a], [Ali 88b]. The advantage of this method is that there is no delay in setting up the environments of the newly spawned processes and evaluation of OR processes can start immediately when the parent ceases. However there are two problems: first as there is no way of predicting in advance whether a process will create offspring or how many there will be, the balance of "slave" to "master" processing elements cannot be accurately judged. Of course in a system where the processing resources are less than the total amount of work to be performed at any one time, queuing of processes within both master and slaves will result in few if any idle processing elements. However the same is true for the system implemented in this project: most receiving processing elements will not be idle during packet transmission as they will be executing a previously received process held in their local execution queue.

The second disadvantage of the "broadcast-while-processing" approach is that for a reconfigurable system such as this, the broadcast bus will be tied up in use for considerably longer periods, thus causing contention for the busses. This leads to processing being blocked as processing and broadcasting would be coupled operations.

6.5.3. The Processing Elements

The evaluation of processes takes place within the individual processing elements of the system. The hardware for each element needs to support input and output from and to the multiple bus system, the storage of rules and the processing of packets. The outline of the proposed design is shown in Fig.6.9. The description given here represents a summary of the section in [Brown 89].

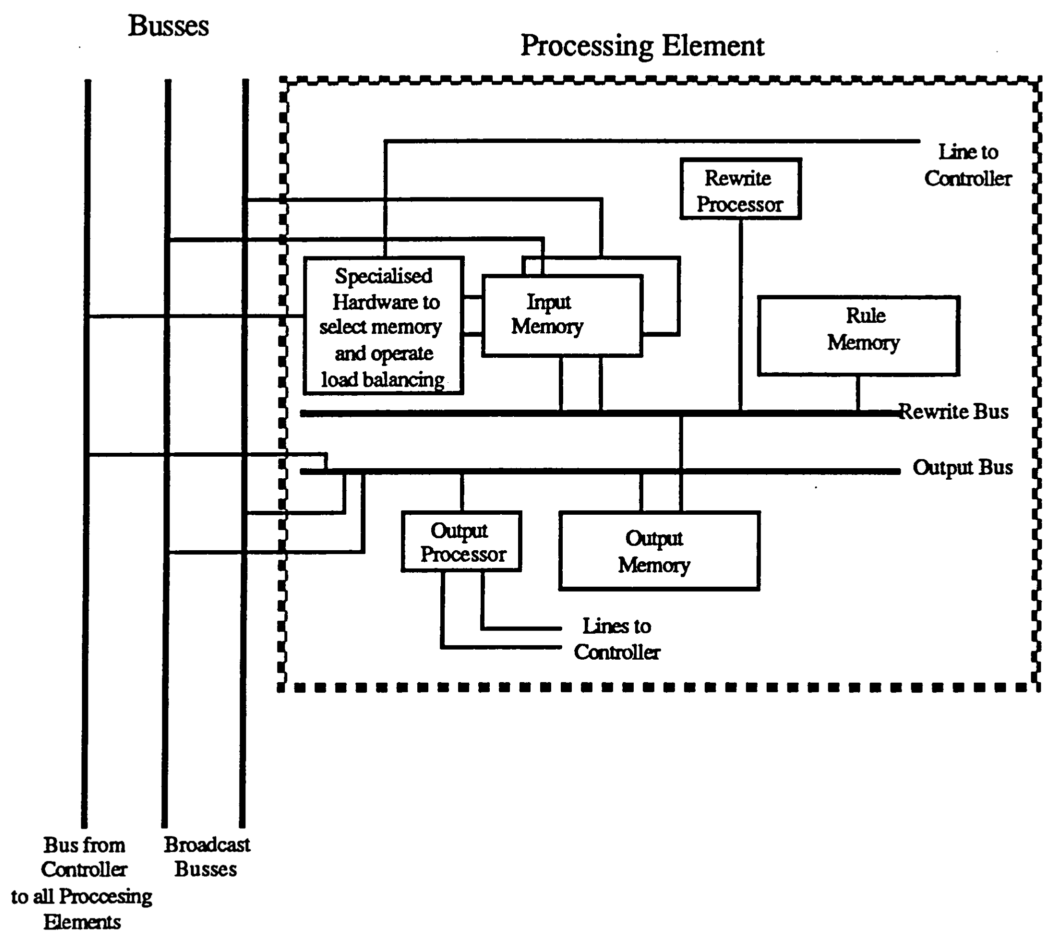


Fig. 6.9 - Outline Design of Processing Element

The unit consists of two processors and a number of designated memory units. The rewrite processor is responsible for process evaluation, and the second processor handles outgoing broadcasts. The static rewrite rules are stored in the rules memory. There are multiple input memories, one for each bus, but only one output memory.

Each input memory holds a queue of packets received from its bus. Rather than utilise a third type of memory the proposal is that the rewrite processor performs the rule rewriting operations in the output memory, destroying the results in the event of failure. The second or "output" processor is responsible for broadcasting the packets produced by the rewrite processor in the output memory when a bus becomes available.

This implies that there are two potentially simultaneous operations on both the input and output memories: for the input memory data may be written to it from the bus at the same time as the rewrite processor is reading from it, and the output memory may be read by the output processor while the rewrite processor is writing to it. These operations can be achieved by the use of memories with separate input and output ports.

The proposal is that the rewriting of processes should take place in a slightly different form from that defined in the abstract interpreter. In many cases a considerable portion of the incoming data packet has to be reconstructed in due course as part of the outgoing packet representing the spawned child processes. In order to avoid this decoding, copying and reassembly it is suggested that the data items in the incoming packet that are not changed in rewriting remain in the input memory: the rewrite processor only writes altered data structures to the output memory, and marks with pointers to the input memory locations the original values which are still valid. The output processor should construct the outgoing packet from the data held in both the input and output memories. The rewrite processor although reading from the input memory cannot be allowed to write to it as broadcasting, ie writing to an input memory, may take place at any time and cannot be delayed. Thus the data in the output memory represents the changes made during rewriting to a data packet which is held in one of the input memories.

The output processor's task is to construct and broadcast data packets as they become available for broadcasting. In the situation where a bus is immediately available the operation will take place with little delay after rewriting has finished. The request for a bus could be made either by a shared bus or a separate single line for each processing element; the best arrangement has yet to be determined. The number of processing elements needed as recipients of a broadcast is known from the disjunction and is stored with the data packet. The allocation and identification of a bus from the controller is supplied over a single conventional addressable bus and with this information the output processor initiates transmission from its output bus onto the broadcast bus indicated.

The question of task scheduling at the level of the processing element is interesting. On the macro level the controller has the job of ensuring an even spread of work throughout the processing elements, but internally for a processing element, work is represented by data packets sitting in the input memories. The priority in job scheduling within the processing element is to ensure that none of the input memories overflow and maintain an even spread of work throughout them. The timing and amount of data received for each input memory is out of the control of the local processing elements, and hence the rewrite processor simply has to take packets from the fullest input memory, at the same time avoiding any memory that is also involved in a broadcast from the output processor as this means that contention will occur as both rewrite and output processors will be reading from the same input memory. If there is no contention for processing resources the evaluation of the solution tree for a given query takes place in a breadth first manner (see Chapter 4.3.2) However this second level of process scheduling as well as the load balancing operations in the controller may lead to an unpredictable pattern of exploration of the solution tree. This does not matter if the performance criterion of the machine is to obtain the full set of answers to a query as quickly as possible. On the other hand it may be desirable to tune the system to produce the first answer in the shortest time by moving to a more depth first approach, and in this situation the two independent scheduling operations will make this more difficult.

One of the aims in developing the simulation was to obtain data about the predicted usage of the input memories during query evaluation. The macro scheduling of work, ie designating processing to processing elements based on a measure of their overall work load, has been modelled in the simulation. At present the internal scheduling of process evaluation depending on which input memory has the highest number of waiting processes has not been implemented.

6.5.4. The Controller

The controller performs two crucial functions: the allocation and configuration of busses, and the designation of processing elements as recipients of data packets.

The possibility of configuring busses in advance has been raised in connection with concurrent copying of the parental environment into processing elements set aside for offspring processes, and has been dismissed as impractical for this system. It is therefore necessary for the controller to allocate a bus at the time a request is made for a broadcast. As far as the macro system is concerned there is no difference between busses and hence the controller can allocate any free bus (or the first one to become free) for a broadcast. However as has been shown because each processing element has an input memory corresponding to each bus it is important to use the busses in such a manner as to achieve an even spread of data packets in the input memories. It is difficult to envisage a practical system that gives a precise measure of the usage of each input memory in each processing element throughout query evaluation, and hence the simplest method for the controller to use is to allocate busses on a round robin or "least-recently-used" basis. This can be achieved by using a queuing system in the controller and will provide a reasonable measure of load balancing with minimum delays in bus allocation. Information from the simulation should reveal whether this method is satisfactory.

The question of organising work allocation to the processing elements is a complicated one. Because of the nature of the rewriting process it is impossible to predict how long an individual process is going to take to execute. If all processes were of similar computational complexity a simple measure of the number of processes awaiting execution in each processing element would allow accurate load balancing to be performed. It is hoped that results from the simulation will allow this scheduling method and others to be compared. It has been suggested that a better measure of the length of the time a process will take to execute can be derived from inspection of the size of the data packet which initiates it. In either case the hardware of the controller has to store data on the number or size of processes waiting execution in each processing element, and this information needs to be updated at frequent intervals. Hardware to perform ranking of work loads in processing elements has been designed

[Brown 89] and the estimate is that this will allow the set of least busy processing elements to be identified in a time of 350*n nanosecs, where n is the number of processing elements required for a broadcast.

6.5.5. Communication Estimates

Having considered the method by which the three sets of functional units in the multiprocessor machine cooperate to organise the transmission of data it is now possible to give some estimates about the total time communications will take.

The request from a processing element to the controller for the broadcast of a packet to n processing elements results in two operations within the controller: first the decision on which bus to allocate, and secondly identification of the n least busy processing elements. The first operation checks on the queue of bus usage and is trivial in comparison with the second operation which involves the ranking of "busy-ness" hardware and takes approximately 350*n nanosecs. The next stage which can overlap with the ranking process is the serial signalling to each designated processing element that a broadcast is to take place on a given bus. This signal also passes the data on which branch of the disjunction that the processing element is responsible for, in the form of an integer value. This serial process will take approximately 150*n nanosecs for n processing elements but can occur concurrently with the load balancing operation.

Finally broadcasting of the packet takes place. The maximum data transmission rate is determined by the speed at which the recipient processing nodes can accept the data, and this is believed to be in the region of 150-200 nanosecs per word in the data packet, ie total broadcast time is 200*m nanosecs, where m is the packet size. This gives an overall estimated broadcasting time of (350*n + 200*m) nanosecs for an individual packet. (The simulation figures are based on the conservative estimate of the process of (500*n + 250*m) nanosecs for data packet transfer).

The question of timing of processing is treated in the next chapter on the simulation. It has been possible to obtain measured timings for process evaluation as defined by the parallel interpreter. The relationship between these timings and the communication estimates is discussed in Chapters 8 and 9 where the results of various test runs are analysed.

6.5.6. Base Predicate Storage

The current model for the implementation of the parallel PLL assumes the duplicated storage of user defined rules including base predicates in the local memory of each processing element. Whereas it is feasible to consider a real system which holds copies of high level user defined rules, ie the user's program, local duplication of base predicates is unrealistic.

In general base predicates are likely to be stored on disk and retrieved as required during query evaluation. This opens a whole area for further investigation and involves the indexing of predicates for speedy identification and the timing of retrieval from disk. Under some circumstances it may be possible to cut down on delays due to disk access by fetching data in advance of processing: this requires some mechanism to enable predictions to be made about which base predicates are likely to be involved.

Another aspect of disk storage is the inclusion of multiple or parallel disk units which are accessed by the individual processing elements by some form of connection network. The communication paths through this network need to be defined: they can be dedicated channels or reconfigurable in the manner of the communication system between processing elements. Recent research into a parallel database system using efficient parallel access to multiple disk units has shown that considerable performance benefits can be achieved in this manner [Gray 90b].

The importance of base predicate storage and retrieval has been recognised from the outset of this project but no detailed proposals are available at this stage. In order to obtain the optimum system for the parallel PLL architecture a survey of current work on base predicate handling in large Prolog database systems would need to be carried out.

6.5.7. Multitasking

Throughout the description of the proposed multiprocessor system reference has been made to query evaluation as a single task. However it would be naive to imagine that a large multiprocessor system involving

hundreds of processing elements and multiple connection networks would be dedicated to a single user. The design of the machine makes no assumptions about the number of concurrent, independent operations that should be supported. In functional terms the multitasking or multiuser situation can be simulated by the input of a query containing several unrelated disjunctions, eg
(aunt(a "bob") or (b=(365*(sqrt17))) or (route(c "london" "sheffield"))?

The method of process evaluation and spawning will ensure that the three disjoined expressions will be handled as independent queries. In practical terms the only modification to the design of the architecture is to ensure that values returned to the controller as the result of query evaluation indicate to which query they relate. The details of the final communication of results have not yet been completed. To some extent this depends on the simulation results: if it is found that there is considerable contention and delays in returning results on a common bus shared between the processing elements, it may prove necessary to provide more than one results pathway.

6.6. Summary

This chapter has looked at the architectural considerations for a multiprocessor machine suitable for implementing a parallel PLL system. A bus based broadcast architecture has been proposed as matching the functional requirements of the system, and the adaptation of the abstract PLL interpreter to meet this design has been described. A possible hardware realisation for this architecture has been looked at in outline, and this proposal forms the basis for the simulation of the parallel PLL system which is discussed in the next Chapter.

Chapter Seven

The Simulation of the System

7.1. Introduction

This chapter describes the development of the simulation of the proposed hardware architecture. This represents the third aspect of the Parallel PLL system. The other functional units, namely the parallel PLL interpreter and the software to map the parallel interpreter onto a single processor system have been described in Chapter 5. The aim throughout the software development has been to maintain conceptual separation between these three aspects although the whole system is modelled in one program.

The simulation development occurred concurrently with that of the parallel interpreter; this has proved of benefit to the system as it has been possible to design the data structures within the interpreter, eg process structures, to include the control information required by the machine simulation software. In this way the information interface between the functional units has been implemented.

7.2. The Role of Simulation in the Design Process

7.2.1. Model Formation and Evaluation

Neelamkavil defines simulation as "the process of imitating the important aspects of the behaviour of the system .. by constructing and experimenting with a model of the system [Neelamkavil 87]. The initial process of model formation involves the abstraction of the important features of the system under consideration. It may be an already existing system, or one in the process of design as in the case of this project.

Once a suitable model has been produced, the next step involves decisions about the testing of the model. This can generally be achieved in three ways: analytical methods, model simulation or realisation of the model. In the development of a new and complicated system such as a multiprocessor computer the third option, ie the building of the actual hardware, is likely to prove far too costly to be considered in the first instance. Only when promising results are produced by analytical and/or simulation methods will the construction of a prototype be a practical proposition.

The decision about the use of analytical methods or simulation construction will depend on the system under consideration. In practise both methods are likely to be used: an analytical model that exactly fits the system model description will clearly produce better results and make verification easier. However in complex systems, analysis methods will normally involve making simplifying assumptions, and at some stage these may prove unacceptable. At this point the effort in developing the analytical methods to incorporate further complexities is likely to outweigh the work involved in the development of a simulation [MacDougall 87].

Thus the typical design process involves the model abstraction followed by an analysis of a simplified model using recognised techniques. Given indications of satisfactory behaviour, a simulation of the model is implemented and finally if the simulation results are consistent with those of analytical phase, a hardware realisation is made.

This design process is well illustrated in the recent paper describing the development of the PEPSys system at ECRC. As described in Chapter 3.1.2.3 results are presented for an analysis performed to identify potential parallelism within the language, followed by an abstract machine instruction level simulation and finally a multiprocessor implementation. Each evaluation system "plays a part in the overall performance analysis" of the language system [Chassin de Kergommeaux 89].

7.2.2. Simulation Design

Having decided that a simulation of the design model will provide useful information the question of design of the simulation arises. Although none of the previous discussion on the role of simulation has made any assumptions about the method of its implementation, the advantages of using a computer for this task has resulted in the growth in interest in computer modelling and simulation over the past twenty years. Computer models are used for many systems, both natural and artificial, where the complexity of interactions make the problem of formal analysis too unwieldy.

The considerations involved in the design of the simulation program include theoretical issues of maintaining closeness of fit between the model and the simulation structure, decisions about the employment of a specialised simulation language as opposed to a general purpose programming language and the extent to which the simulation should incorporate the results of analytical evaluation [Neelamkavil 87].

7.3. The Requirements of the Parallel System Simulation

It was established at an early stage in the development of the OR parallel PLL system that a simulation of the hardware design would be necessary to produce the type of results that were needed to assess the design. As discussed in Chapter 5 initial analysis of the language and the proposed applications area had shown that there existed the potential for large amounts of parallel computation where alternative solutions to queries were sought. As has been shown in the discussion of parallel logic language systems most other proposals in this area have also recognised the importance of the inclusion of OR parallel execution (see Chapter 3, Fig.3.1).

The information required in order to evaluate the parallel system model was identified as involving two parts of the system: the parallel interpreter and the multiprocessor machine. From the interpreter it was necessary to obtain data on:

a) the number of processes involved in a query and more specifically whether they were spawning or non spawning, ie terminal processes; also the proportion of terminal processes which produced positive responses and those which gave the FALSE result,

b) timings within each process, ie the time spent in rewriting as opposed to converting the data packet into a rewritable expression and in the case of a spawning process, setting up a new data packet for transmission to offspring.

The feasibility of the system may depend on the limitation of the overheads of spawning so accurate timings for these overheads must be obtained.

From the model of the multiprocessor machine data was needed on:
a) the number of busses and processing elements in use throughout the query,
b) delays in processing due to contention for busses or uneven load balancing within processing elements,
c) the amount of storage required in the input memories in each processing element corresponding to each bus.
d) the pattern of return of results during query evaluation.

It was important to be able to collect the above data from a range of different machine configurations, as one of the primary aims in developing a simulation was to obtain information on the optimum ratio of processing elements to busses. Similarly there are a number of possible load balancing algorithms to control the allocation of work to individual processors. It was hoped to use the simulation to obtain information on the best method to use.

The data required from the parallel interpreter could be obtained without a full system simulation but the machine performance predictions required a proper mapping of the interpreter to an architectural simulation. It was recognised that the simulation had to be flexible enough to incorporate supplementary requirements at a later stage. This indicated that the best approach to the design was the traditional software engineering one of top-down development with the software data structures and functions matching the design model as closely as possible.

7.4. The Parallel System Simulation Design

7.4.1. Introduction

The intention of the simulation was to show how the parallel PLL interpreter could be used in conjunction with the bus based multiprocessor machine and obtain data on the performance of the system. The system design involves holding a copy of the rewrite interpreter at each processing element in the system (user defined rules being available to each processing element as well). When processes spawn offspring OR processes these are set up in remote processing elements by the method of broadcasting a data packet to a designated set of processing elements by means of a bus. The controller is responsible for designating the broadcast bus and the receiving processing elements. The essential job of the simulation is thus to emulate the movement of processes round the parallel machine and execute them by invoking the interpreter code. In order to achieve this new data structures representing the machine are needed as well as forming the interface with the structures being used by the interpreter.

The data structure which is used in the interpreter to represent a process is described in Chapter 5.4.3. It holds two types of information: data relating to the rules and query (the process description) and control data giving the process identification number, creation time etc. As far as the interpreter is concerned the data of importance is the process description, it is only the requirements of modelling the system on a single processor that lead to the introduction of the control information in the first place. However with the additional layer of simulation, ie the multiprocessor machine structures, the control information held in the process structure is to be used as a link between the different software aspects. In order to model the interpreter on a single processor it was necessary to maintain a global queue of processes awaiting execution and to execute these processes according to some scheduling algorithm. As far as the interpreter was concerned the scheduling algorithm was unimportant: all processes are independent and all processes must be executed in order to obtain a full set of results. Hence they can be executed in any order as long as the system remains as the abstract interpreter model. However this does not hold true for the development of the hardware simulation.

The point of contact between the parallel interpreter and the machine simulation has to be the scheduling of these processes on the execution queue: in a parallel machine many of them may be running concurrently and the system has to emulate this. Thus the control information in the process structures now becomes of real importance in modelling when a process is actually executed in the parallel system. The machine simulation has been implemented by the inclusion of a module representing the

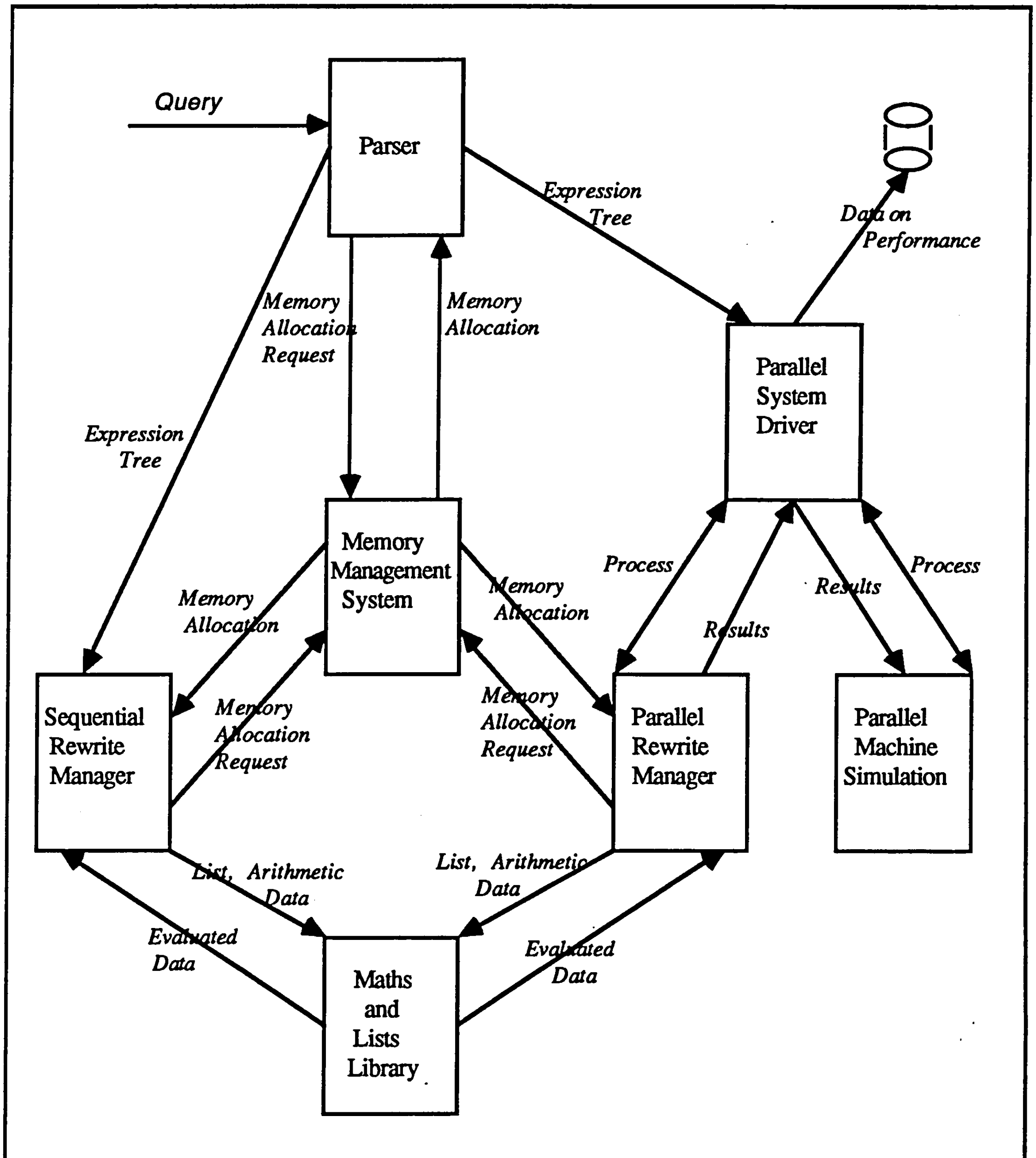


Fig. 7.1 - Data Flow between Parallel PLL System Modules

structures and functioning of the proposed machine. The top level parallel system driver acts as coordinator between the machine simulation functions and the parallel rewrite system, and it incorporates the global information on processes awaiting attention. Fig.7.1 shows the relationship of the various software modules to each other by specifying the data flow between them. The sequential rewrite rule manager has been included in the program: this has proved helpful during program development when checking for consistency of results between the parallel and sequential modes of execution.

7.4.2. Machine Data Structures

It was clear that each processing element acts as a store of information that altered during query evaluation, and in a similar fashion the controller has to maintain a data store if it were to perform its allocation of busses and load balancing tasks. The basic data structures to represent the parallel machine were therefore defined as:

a) a controller

b) an array of processing elements.

Fig.7.2 shows these data structures and indicates the data relationships between them. It indicates how the separation between the machine structures and the interpreter is bridged by the two global queues, ready_to_run_queue and ready_to_allocate_queue. This will be discussed in detail below.

The controller data structure was subdivided to hold data on:

a) the amount of work each processing element was performing throughout the execution time; this was represented by a count of the number of processes awaiting execution in each processing element at any given time,

b) the timings of processes on each processing element; data was held on the finish time of the latest processes to run on an individual processing element, ie it represented the time at which a processing element became available to execute a new process,

c) data on busses in use throughout the execution time.

These internal arrays involved the storage and updating of timings during each run. The decision to represent work loading as a simple count of processes waiting execution was made as a first approximation: the suggested measure of work load involved looking at the "size" of each process waiting for execution (see Chapter 6.5.4).The load balancing methods are discussed in Chapter 9 in light of the results obtained from the simulation.

Each member of the array of processing elements held data on:

a) processes awaiting execution in that processing element,

b) processes that had been spawned within the processing element and were awaiting allocation and transfer to remote processing elements,

c) the maximum usage value reached as execution proceeds for the input memory corresponding to each bus,

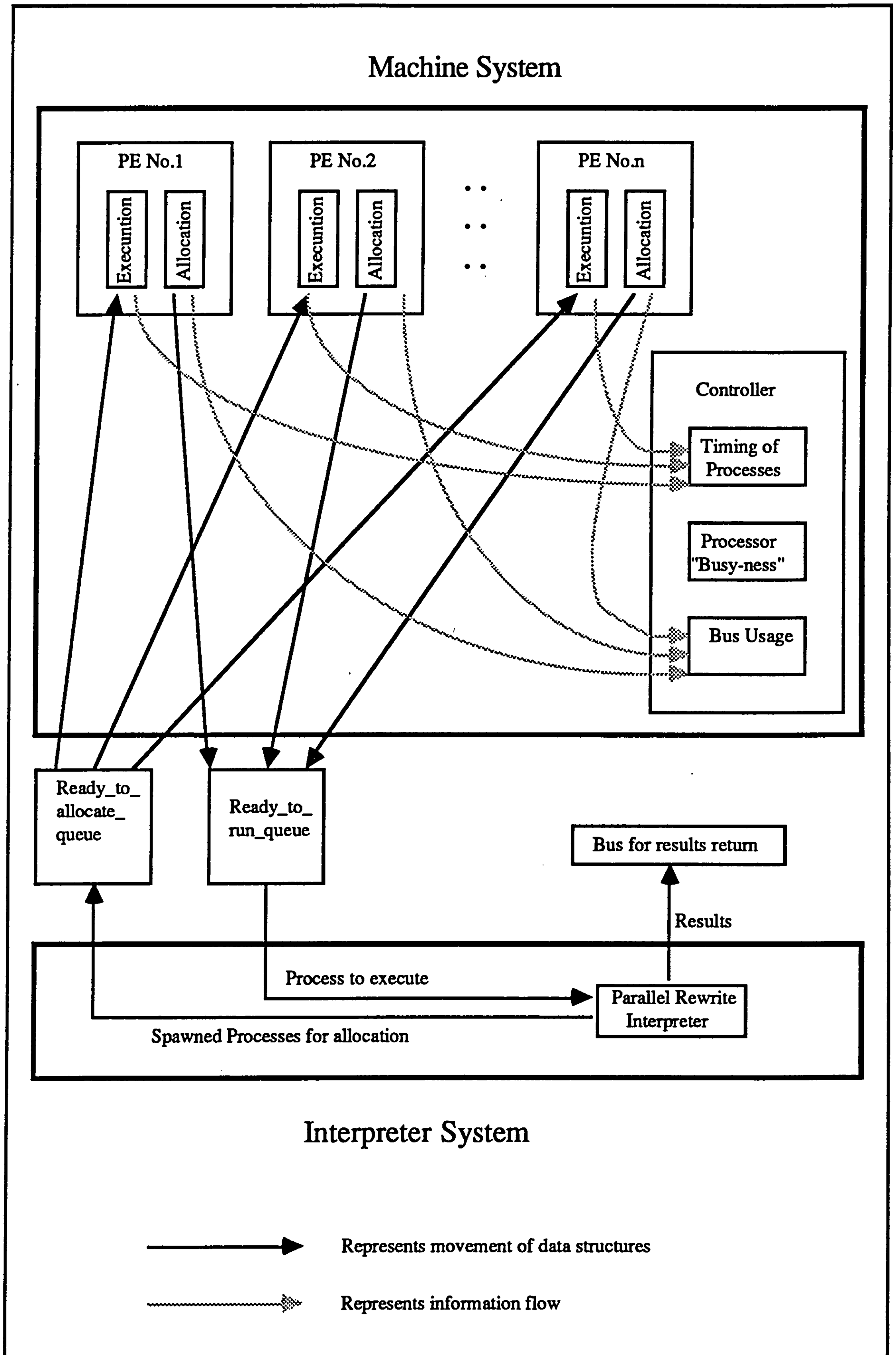


Fig. 7.2 - Machine Data Structures

d) the time that the maximum input memory usage was first reached.

The first two data items, ie the process queues, consisted of a linked list of process structures which were defined in the format shown in Fig.7.3.

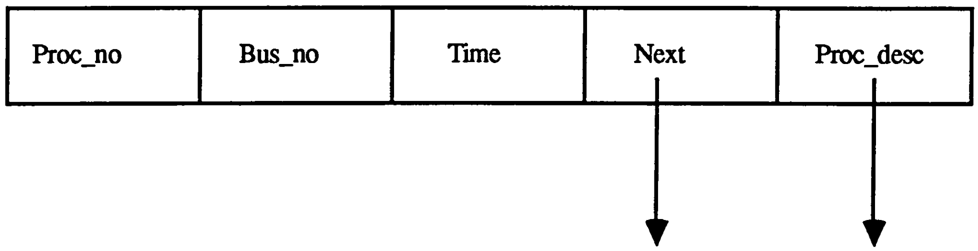


Fig. 7.3 - Process Representation

Proc_no represents a unique global identifier, Bus_no refers to the bus identifier on which the process was transmitted to the processing element and Proc_desc is a pointer indicating the list which defines the expression to be evaluated (see Chapter 5.4.3). Time means creation (spawn) time in the case of processes awaiting allocation, and time of reception at the designated processing element in the case of processes waiting to execute. In the case of a process that was transmitted as soon as it was spawned (ie no delay due to bus contention) the Time field would be incremented by the message passing time when it is moved from the parent processing element to its designated processing element. Obviously if there was a delay in obtaining a bus this delay would have to be added to the new Time value.

It can be seen from this description that processes awaiting execution now reside within an individual processing element structure rather than on the global queue of the abstract interpreter. In fact, to ease processing, the global queue was maintained and augmented with a similar queue representing processes awaiting allocation. These global queues now held modified structures known as "process records" and "allocation records" respectively; these did not include any of the process description or bus details but merely acted as a central list of where to find the appropriate process and its time value for scheduling purposes. Process records were defined as shown in Fig.7.4.

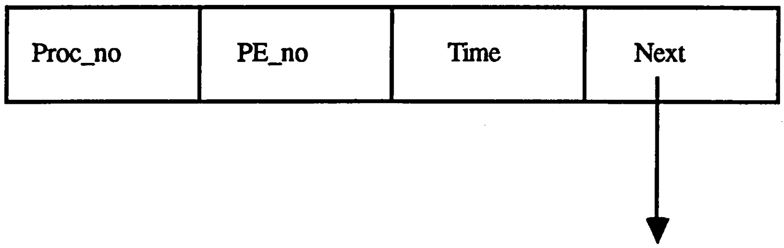


Fig. 7.4 - Process Record Representation

Allocation records were slightly different in format because they each represented a group of processes that had resulted from a spawning operation in the parent process. They were defined in Fig.7.5.

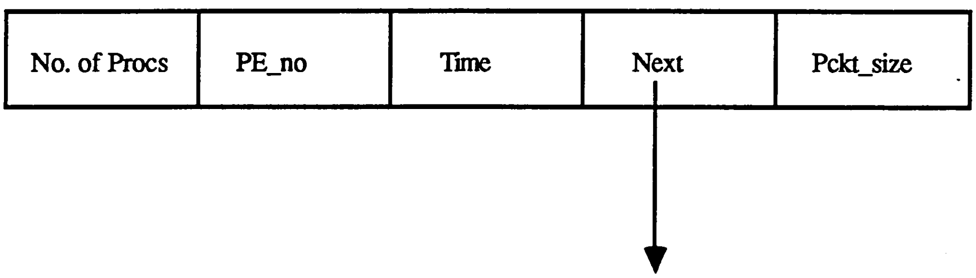


Fig. 7.5 - Allocation Record Representation

Instead of using one allocation record per process this combined record showed the number of processes spawned in the first field. This data plus the processing element identifier and the time of spawning allowed the group of processes being held on the allocation queue within the processing element to be identified. The fifth field held data on the size of the data packet which represented the group of processes and would be transmitted to the receiving processing elements (see Chapter 7.4.3.3).

The two global queues holding them were designated "ready_to_run_queue" and "ready_to_allocate_queue". Unlike the data structures representing the controller, the processing elements and processes, these global queues would have no direct counterpart in the real system.

7.4.3. Functional Design of the Simulation

7.4.3.1. Introduction

The design model of the system required that each processing element held a copy of the parallel interpreter including the user defined rewrite rules. To include multiple copies of the interpreter code in the simulation program was not necessary and would have involved a great waste of space resources, thus one copy of the interpreter code was used in the program but with an extra parameter (the processing element number) to show the processing element in which it was operating at any one time. Similarly the arrangement developed for the abstract interpreter that the general system stack was used by each process in turn and then subjected to garbage collection, was maintained as a representation of the local evaluation stack within an individual processing element.

The top level functional task of the simulation was to model the movement of processes round the machine and execute them according to a predefined scheduling policy. In the real machine many of these events, eg the execution of processes, the broadcasting of data packets, would take place simultaneously and the task of the simulation was to model this behaviour in a sequential fashion. The manner in which timing values were arrived at is described below, numerical values being introduced for:
a) the execution time of each process,
b) the transmission time of a data packet on a bus,
c) delays due to non-availability of a bus,
d) delays due to lack of idle processing elements.
c) and d) represent queuing problems and one approach would have been to employ a recognised queuing theory method to simulate this. However the decision was made to attempt to provide more accurate timing values based on measurements and estimates calculated from run time observations of process execution. Nevertheless this approach does involve the introduction of approximations and there are computational overheads to be considered. This is discussed in Chapter 7.4.3.3.

7.4.3.2. The Prototype Simulation

The conceptual separation between the hardware simulation and the parallel interpreter has been emphasised and this allowed the first version of the simulation to be set up using dummy processes. In this version instead of calling the proper interpreter to execute processes according to the rewrite rules, a piece of code was inserted to model this. It read in a process and either terminated without spawning or spawned a random number of processes. As it created these new dummy processes (which had no process description component) it allocated a creation time to the offspring group based on the creation time of the parent plus a randomly produced parental execution time. Similar random timing values were used to represent bus transmission times.

The top level algorithm that was initially applied to this system was:

```
while (process records on ready_to_run_queue)
   {choose earliest process record on the queue,
    call the interpreter for process corresponding to the process record,
    update all relevant queues,
     if (allocation record on ready_to_allocate_queue)
      {distribute the corresponding processes,
       update all relevant queues
      }
   }
```

There were a considerable number of refinements that were considered at this stage and some of them implemented. For example under one process allocation strategy, the first process spawned by the interpreter was automatically given to the same processing element as its parent regardless of its work load.

The first version of the simulation employed a simplified method of representing concurrent execution: although this system produced information on the various individual components during program execution these were subject to a number of approximations. This is discussed fully in the context of the full system version (Chapter 7.4.3.4).

The approach of setting up the machine simulation using simplified processes was useful on two counts: first it allowed a great many small data manipulation and checking functions to be installed and tested in a simple system, and secondly it provided a good test bed for the development of the top level algorithms to model the architecture.

However it became obvious that in order to test the system under proper working conditions real data regarding processing times, data packet size etc was required. This necessitated a move to incorporate the full rewrite interpreter into the simulation and to construct functions to measure or evaluate the necessary timings.

7.4.3.3. Timing Data

In order to predict the performance of the parallel machine model it was necessary to include timings for two aspects of the functioning:
a) the length of time each process takes to execute,
b) the length of time a data packet takes to move from one processing element to the receiving elements.

If these two sets of timings were known all the other behavioural aspects of the system, such as traffic on the busses, work load on each processing element etc, could be calculated. In addition it has been stated as one of the simulation requirements that the process execution times should be capable of differentiation into "true" processing, ie rewriting, and data packet construction and decomposition in order to check that this form of process spawning was not creating unacceptable overheads.

Measurements of actual process execution times proved impossible with the initial resources available. The original computer used for the development of the system was the Sun 3/60 workstation, and attempts to use the system clock in order to measure process times ended in failure as the granularity of the clock was too coarse. It would only measure in intervals of approximately 16 ms and typical process times were considerably smaller than this.

In order to obtain as accurate estimates of process times as possible Assembler listings of the C interpreter code were obtained. The task of enumerating and summing the clock cycles taken by each function was performed and these values were inserted into the interpreter code. Thus whenever an interpreter function was called the timing count for that function was incremented.

The first version of the full simulation relied on these calculated process timings. However two sources of inaccuracy were identified: first the sheer laborious nature of the Assembler inspection task must have led to human errors being made, and secondly full data for the MC68020 processor (as used by the Sun 3/60) was not available and thus timings were based on the instruction set for the MC68000.

At this stage in the project it was decided to experiment with porting the system to a Transputer based system, namely an IBM AT clone with a Transputer card holding a T414b-15 and 2 Mbytes of memory [INMOS 89]. The 3L C compiler for the Transputer had recently become available and this was used to recompile the parallel system software [3L Parallel C 88]. The main advantage of this was the prospect of using the Transputer system clock which operates at a granularity of 1 microsec. The additional clock reading instructions were inserted in place of the calculated measurements and all results presented in Chapter 8 refer to this version of the software. The estimated timings based on the Assembler inspection for the Sun PLL system have not been used but the work involved in the task provided additional information into the detailed operation of the rewrite interpreter, eg the computational overheads because of function calling and recursion, which are considered in detail in Chapter 9.

The second type of timing data that was needed by the simulation was the length of time a data packet took to transfer on a bus from one processing element to another. This value had to take into account the setting up time for a bus as well as data transfer. Any delay due to contention for a bus had to be quantified and added. Obviously there were no actual measurements that could be made by the system clock to obtain these figures unlike the data on execution times. The timing data for packet transfer therefore had to be calculated.

It has been shown in Chapter 5 that the time of packet transfer depended on its size and the number of receiving processing elements. However as has been discussed the software did not actually construct the data packet using instead the movement of process structures to model the information flows in the system. At this stage rather than alter the model to work on the basis of transfer of "real" data packets, additional functions were added to calculate the size of a data packet using information from the group of processes that it would represent in a true realisation of the design.

In Chapter 6 the details of the data items in the data packet are discussed. The tagging of each item with a three bit tag has been assumed, giving data item sizes ranging from eight bits for a user variable to nineteen bits for an integer or floating point number. The introduced variables and pointers were designed with sixteen bit representations. In order to simplify the calculations it was decided to use the figure of sixteen bits to express the size of any data item in the packet. It was important that the size of the data packet should not be underestimated but it was felt that this approximation was unlikely to do this. Thus the size of the data packet was calculated and as the number of receiving processing elements was known it was possible to include the time of transfer of an individual packet. Timings used for the passage of the data packet were based on the estimates given in Chapter 6.5.5, ie the time to complete a broadcast was calculated at (500*n + 250*m) nanosecs, where n is the number of recipient processing elements and m is the word size of the data packet. Delays due to bus contention were known from the stored information within the controller on bus utilisation.

7.4.3.4. A Better Representation of Concurrency

The main simplification in the first process scheduling algorithm was the assumption that if the processing followed the pattern of "execute earliest process then allocate spawned processes" that this would simulate the behaviour of the concurrent system. In fact it only provides an approximation of it: the reason for this lies in the allocation strategy.

In order to achieve the optimum sharing out of work in the parallel machine, it is necessary to know at the time of allocation of a process the work load that exists in all the processing elements at that time, so that the process can be sent to the least busy. With process execution times of variable and unpredictable length it is not possible to update the state of all the processing elements accurately at the end of executing one process under the first scheduling algorithm.

Consider the situation shown in Fig.7.6: at time T1 the process on PE no.1 will be chosen for execution by the simulation software finishing at time T2. A review of processes queued up in the different processing elements performed at T1 will show correctly that PE nos.2, 3 and 4 have one process each. However a review made at time T2 following the

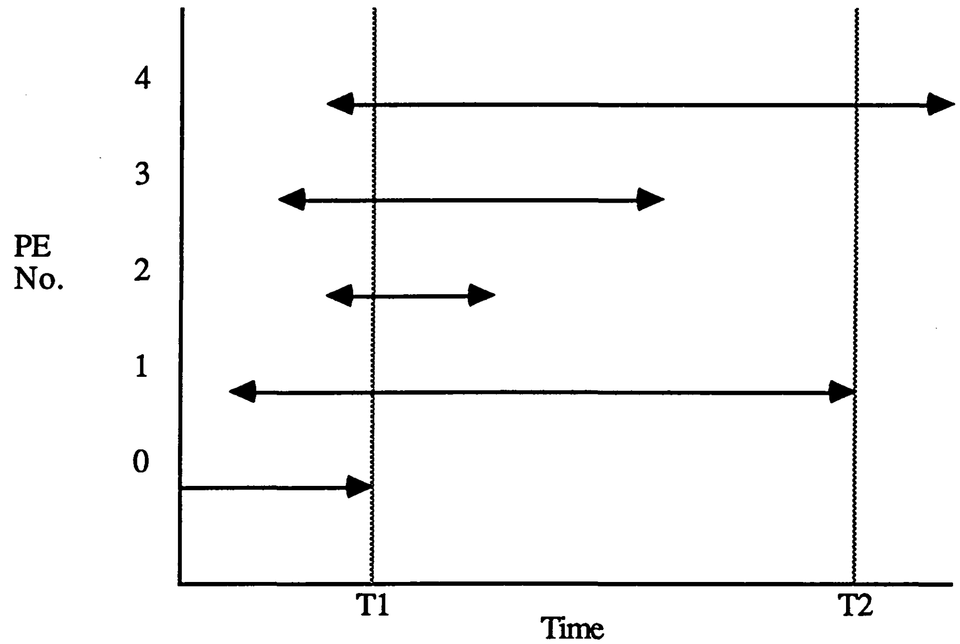


Fig. 7.6 - Timing of Process Execution

execution of the process on PE no.1 would not be able to predict that PE nos.2 and 3 had completed the execution of their processes because they were substantially shorter that the processes on PE no.1. It would however be able to judge correctly that PE no.4 was now involved in executing its process by inspection of the Time field in the process. Thus the information held in the controller about the state of work load of each processing element is likely to include some inaccuracies if this method of simulating the scheduling of processes is maintained. The degree to which these inaccuracies may effect performance figures is difficult to quantify and will vary from query to query. However as one of the important aims of the simulation was to test different load balancing strategies it was important to try to avoid inaccuracies in this area. For this reason a move to a more realistic approach towards the modelling of concurrent process execution and allocation was attempted.

The first approach to the representation of concurrency in the simulation can be described as event orientated or the variable time method. In this the state of the system was checked at time intervals set by the start and finish of a particular executing process. As shown above the use of this approach with the single global queue of processes ready to execute led to inaccuracies in the information about the work load in individual processing elements. Two corrections were possible for this situation: the first involved halting at the end of a process execution, and at that stage performing a trial execution of all processes that could interfere with the outcome of it (ie affect the work load of other processing elements in the case of the first process having produced spawned processes). The processes which were executed to obtain information would then either have to be subject to cancellation or roll-back, or their results stored on a "future" results list. This system was rejected on the grounds that it could involve considerable extra storage.

The second method of time representation is that of interval orientated simulation which involves stepping through the system at fixed time intervals and operating the system at that point. A variant of this was used for the full version of the simulation. Fixed intervals updating of the system data was used, but as the system model had been developed the execution of an individual process was an atomic, ie indivisible operation, and there was no attempt to alter this or halt execution of a process midstream. These considerations led to the development of the following algorithm for the "time step" method:

```
set System_time to Time_step,
while there is still processing to do
  {while (process_records on ready_to_run_queue with
                                          Time less than System_time)
        or (allocation_records on ready_to_allocate_queue with
                                          Time less than System_time)
    {while (process_records with Time less than System_time)
     {identify corresponding process,
      execute process,
      update relevant queues
     }
     while (allocation_records with Time less than System_time)
      {identify corresponding processes,
       distribute processes to suitable processing elements,
       update relevant queues
      }
    }
    increment System_time by Time_step
  }
```

This method meant that for each time step in the processing of a query all processes whose Time, ie start time value fell within it were executed before any allocation of spawned processes took place. Data on the duration of each process was stored so that when the software came to allocate all the resulting processes, the full information about the work state of each processing element was available. Of course allocation of processes during the time step could result in further processes becoming executable within the time step, so the loop of "execute all processes then allocate all spawned processes" was repeated until no further action was possible within that time step. The system time was then incremented to the next time step and the entire loop restarted. Fig.7.7 shows the top level functioning of the system under this modified algorithm.

One criticism of this method is that injudicious choice of the time steps leads to large computational overheads: however these take the form of processing time not memory usage, and as far as this simulation was concerned total times for query answering were not excessive, the limiting factor proving to be storage space. This is discussed in more detail in Chapter 8 in the section on benchmark tests but a typical run time for one of the larger queries supported by the system was under three minutes.

7.4.4. The Full Simulation

The full simulation software has been implemented in a suite of interactive modules as shown in Fig.7.1: these comprise the main program, the parser module, the memory management system, the sequential rewrite manager, the parallel rewrite manager, the parallel machine simulation module which includes the parallel system driver, and a small library of mathematical and lists processing functions. The entire system represents more than 200 Kbytes of source code. The modules written specifically for this project are the parallel rewrite manager, details of which can be found in Chapter 5.4, and the machine simulation. This latter module contains the top level parallel system driver as well as the functions simulating the machine operations; it contains over 2000 lines of code and occupies almost 70 Kbytes.

On entry to the PLL environment the user can opt for the "parallel" system; in this case the simulated machine is "configured", ie the user is asked to decide how many processing elements and busses they require

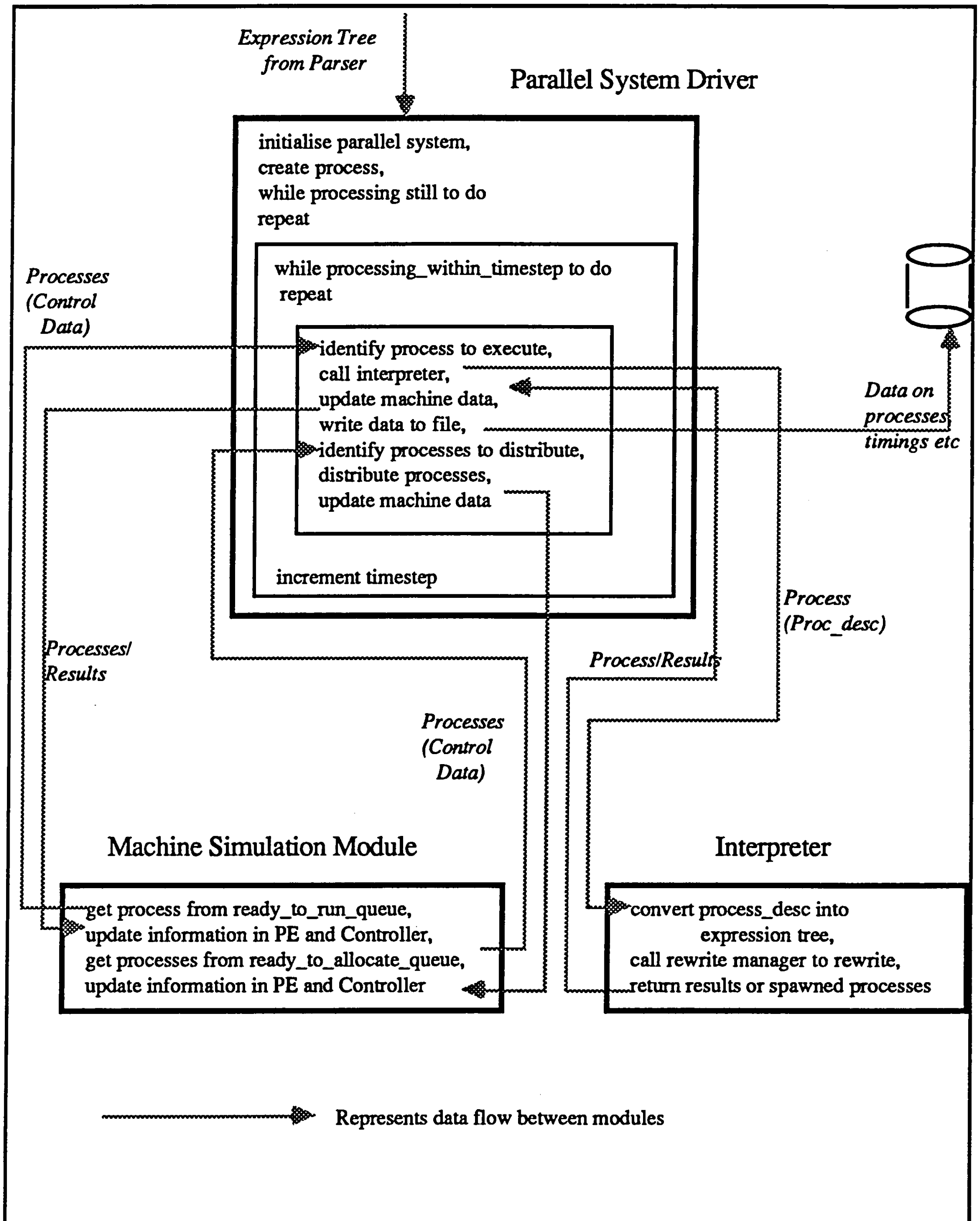


Fig. 7.7 - Functional Design of Simulation Software

for a given run. Various trace options are also presented. The query is then parsed and the expression tree set up on the system stack in the normal way. Control is passed to the machine simulation which sets up a process corresponding to the execution tree, allocating a starting processing element and initialising all the queues and control information. This is shown in Fig.7.7.

The time stepping algorithm is then responsible for the execution of this process as well as any subsequently spawned processes by the manipulation of the information and queues held within the parallel simulation data structures. The rewriting of expressions is accomplished by calling the parallel interpreter from the appropriate module. The high level functioning of the system is presented in the pseudo-code in the previous section and diagrammatically in Fig.7.7 Further details of the actual C functions involved are given in Appendix F.

7.5. Summary

This chapter has described the development of the software which emulated the behaviour of the multiprocessor architecture. This has involved the introduction of data structures to represent the various functional parts of the machine and the storage of control information needed to implement processor work load balancing and bus scheduling. The algorithm used to model concurrent operations in a sequential manner has been presented. The requirements for the system have been discussed, and the manner in which the parallel machine simulation interfaced with the logic interpreter looked at.

Chapter Eight

Preliminary Testing and Results

8.1. Introduction

The testing of the parallel PLL system and the machine simulation involved two separate types of experiments. In order to check that the new interpreter operated its rewrite rules correctly, a series of small programs written in the PLL were used. These tested the new versions of the rules and ensured that the parallel interpreter produced the same results as the original sequential one. By incorporating the two versions of the interpreter into the same suite of programs it was possible to toggle between parallel and sequential execution modes and check consistency in this manner.

These tests were used in the development of the parallel interpreter and included queries containing reference to user defined rules, conjunctions, disjunctions and all the other operations permitted in the PLL. In designing these programs there was no attempt to model the type of application for which the system was designed; their purpose was to confirm the correct operation of the parallel interpreter with respect to the sequential one.

The other programme of testing used the simulation in order to provide behavioural predictions for the system. These benchmark tests are the subject of this and the following chapter. The simulation of the multiprocessor architecture was produced in order to provide information on the type of behaviour to be expected from a hardware implementation of the design. The manner in which the simulation had been written allowed several of the system parameters to be varied, the intention being to obtain data on performance under a range of differing configurations. This information would form part of the machine design process.

In order to obtain maximum value from the simulation it was necessary to identify the aspects of the system design which needed to be tested and to devise a series of benchmark tests to accomplish this.

8.2. Testing of the System Design

Work on the parallel PLL interpreter had shown that it was possible to implement correctly the abstract model for OR parallel process execution based on the use of rewrite rules as the inference mechanism. However the question remained as to the performance benefits that this approach was likely to bring, and in order to obtain quantitative data on this the simulation was prepared. Data was produced by the simulation on two basic aspects of the system, ie the performance of the parallel interpreter and the predicted performance of the multiprocessor machine.

Two parameters of the system definition could be varied in the simulation: these were the configuration of the machine, ie the bus and processing element numbers, and the scheduling algorithm responsible for load balancing between different processing elements. When the simulation was run the machine configuration had to be set by the user for each series of queries. The permitted number of broadcast busses ranged from one to twenty; it was not envisaged that the real machine would have as many as twenty busses but by allowing a high maximum figure it should be possible to obtain information on situations where there is virtually no contention for the communication medium. The range of processing elements was originally two to one thousand. However when the software was transferred to the Transputer based system the maximum was reduced to one hundred. The more limited memory space in this system meant that a balance had to be struck between the space requirements of the simulation software and the space designated for query evaluation. By reducing the maximum number of processing elements to a hundred (and hence reducing the size of several arrays within the simulation code) it was possible to provide approximately 1.4 Mbytes memory for the PLL evaluation stack. Even so this figure was considerably lower than the stack space allocated by the Sun network and resulted in limitations on the size of rule base that could be queried.

The scheduling algorithms for process allocation have been touched on in Chapter 6.5.4. The first problem involved estimation of the work load in each processing element throughout query evaluation. Two suggestions were made for this: a simple count of the number of processes waiting for execution within a processing element, or a count of the total "sizes" of processes awaiting execution, the size of a process being defined as the size of the data packet which inaugurated it. Having obtained a quantitative measure of the work load in each processing element at any particular time, the most straightforward scheduling method was to allocate processes in turn to the least busy processing elements. A variation on this which was also tested was automatically to allocate the first process spawned to its parent processing element, the subsequent ones to the least busy processing elements. The other allocation approach is to take a round robin approach, ie processing elements were chosen on a circular queue method on a least-recently-used basis. Finally as a check that process scheduling is providing benefits, tests were made in which processes were randomly allocated to processing elements.

8.3. Required Results

8.3.1. Introduction

The results to be obtained from the simulation could be divided into two main categories: those representing data on aspects of the performance of the multiprocessor architecture, and data on the behaviour of the rewrite interpreter. It is important to note at this stage that the data on the interpreter was in the form of "real" timings, whereas that on the machine performance was produced by calculations based on the proposed design of the hardware.

8.3.2. Rewrite Interpreter Timings

Data was needed on the number of processes produced during a given query and the time taken for each process to execute. It was also essential to measure the time taken by the different subtasks in process evaluation in order to check that process spawning was not introducing unacceptable overheads. Thus for each process measurements were made for total evaluation time and the following subdivisions of evaluation time:
a) set-up time, the time taken by the interpreter to convert an existing process structure into an expression tree,
b) rewrite time, the time taken to rewrite an expression tree until no further alterations possible, or until an OR node encountered,
c) spawn time, the time measured between first recognition of an OR node and the production of a full set of new process structures ready to be allocated.

8.3.3. Machine Performance Data

Results on machine performance were sought for a range of different machine configurations. The first result to be obtained was a measurement of the total time taken to complete each query. This allowed speedups to be calculated and gave an indication of the effect that contention for the communication medium was having on overall performance.

However in order to understand what was happening in the machine during query evaluation it was necessary to obtain results on the execution and starting times of each process. When these were related to the processing element to which a process was allocated the pattern of machine usage can be determined. With a good scheduling system it was expected that the number of processing elements in use would rapidly increase to the maximum number and that this level would be maintained throughout query evaluation.

Incoming data packets were written into the input memories of a processing element, each input memory serving a particular bus. As discussed in Chapter 6.5.4, the method of allocation of processes to processing elements was by assessment of work load, and bus allocation was performed on a round robin basis. The intuitive belief was that this method of bus allocation should result in a reasonably balanced usage of input memories for each processing element. In order to confirm this, measurements of the size of data packets waiting for execution in each input memory throughout process evaluation were required. The simulation does not manipulate data packets as such, maintaining the original format of the abstract process structure. This has meant that additional functions to relate the calculated data packet size with the input memory have been written and this information updated throughout query evaluation. The final result was a table of maximum input memory usage for each processing element and the time at which the maximum was first reached.

The information required on the transmission of data packets across the communications medium fell into two categories: the bus usage, including any transmission delays because of non availability of a bus, and the size and transmission time for individual data packets.

The question of return of results, ie the correct binding values, to the controller was discussed briefly in Chapter 6.5.7. This could be achieved by use of a common bus shared by all processing elements, or more localised busses used by fewer processing elements. In order to assess which design should be chosen it was important to have data on the pattern of results return. If positive results become available in such a manner as to introduce little contention for a globally shared bus there is no point in constructing a more complicated system. With this in view information was sought on the times at which final bindings were produced by processes, and the size of data packets that these bindings give rise to. The format proposed at present for the results data packet is identical to that of the spawned process data packet, as it is recognised that "results" may not always consist solely of binding values, but in the case of "uncomputable" expressions will include reference to the expressions themselves. More work in this area is indicated but it was hoped that the simulation would provide data on which to base future design decisions.

8.3.4. Results Summary

To summarise, the requirements for results involved timing measurements for each query evaluation run. For each combination of system definition, ie processing element/bus configuration and process scheduling algorithm, records were made for:

a) process start and finish times,
b) within each process, the set-up, rewrite and spawn times,
c) process/processing element allocation,
d) input memory usage levels,
e) data packet sizes and transmission times,
f) bus usage and delays in obtaining busses,
g) results times and results packets sizes.

These results were written to file during query evaluation, and the data analysed in various ways afterwards. This represents a fourth aspect to the software developed during the project: functions were included in the simulation to calculate and output the appropriate data, and a separate data analysis program was written to show the results in a tabular or graphical format. An example of the output of this program is shown in Appendix G.

The intention was that this series of tests and subsequent data analysis would complete the work on the simulation. However as will be shown the results obtained from the first series of tests produced somewhat unexpected results, and led to the decision to alter the testing strategy and include further modifications to the system.

8.4. Benchmark Tests

8.4.1. Requirements in Benchmarking

Williams defines a benchmark as a "program or set of programs which allows the performance of similar system features in different implementations to be compared" [Williams 87b]. In this instance the systems to be compared are the versions of the parallel PLL under differing configurations and the sequential PLL. The question of performance comparison between the PLL and other logic programming systems, notably Prolog, lies outside the scope of this project.

Benchmark tests can either be specifically written for the system under consideration, or can use existing programs. In the case of the PLL because it is a new language system it has been almost inevitable that new programs have had to be written, although some simple test programs were available from ICL with the sequential interpreter. In the main these were used for testing the correct working of the parallel interpreter with respect to the sequential one. Direct translation of Prolog programs into the PLL has been shown to be theoretically sound in the case of "pure" Prolog programs [Cooper 87c], but problems arise when extra-logical features are involved. Nonetheless one of the test programs used for benchmarking has been directly translated from Prolog (see Appendix C).

When designing benchmark tests care must be taken that the following points are covered:
a) that they allow valid comparisons to be made between different systems,
b) that they test the whole system not just certain features,
c) that they are of suitable size,
d) that the area under testing is clearly defined.
The tests developed for the PLL simulation meet some of these criteria as discussed in the following section.

8.4.2. Test Programs

Having considered the criteria for successful definition of benchmark programs it was realised that these would be difficult to achieve in full for the PLL system. The main problem was shortage of memory space: the system had been moved from the Sun workstation to the Transputer board in order to obtain accurate processing timings, but it was recognised that the result of this was a decrease the amount of memory for the PLL evaluation stack.

The new PLL system had been based on the concept of OR parallelism because analysis of the applications area had revealed the potential for this form of parallel execution. It was thus realistic to construct benchmark tests which modelled this type of application. Work has been done for the Alvey program on defining benchmark programs for testing architectures in large knowledge based systems [SIGKME1 87]. This has identified a number of programs using rule based systems and includes the Protein Molecular Structure Database, RESCU real time expert system, OPS5 production rule system as well as the smaller Prolog test programs known as the Stockholm Tests. Many of these programs show potential for OR parallel execution and would have been ideal candidates for benchmark testing of the broadcast bus multiprocessor architecture. However apart from the language compatibility problems, the simulation capability was far too small to consider their use. It was therefore decided to produce suitable PLL benchmarks specifically for testing this system.

Two different sets of user defined rules were written: the first based on the family data base concept, and the latter a direct translation of Pereira's map colouring problem [Conery 85], [Campbell 84]. Full details of the programs and the queries used are given in Appendix C. It can be seen from these that it was not possible due to space restrictions to run some of the queries with all variables uninstantiated; the query to the map colouring program:
colour(v w x y z)?
which generated over 7000 processes on the Sun system, ran out of space on the Transputer. With one variable instantiated the query:
colour("red" w x y z)?
produced 1885 processes and gave the full set of bindings on both systems.

The testing of the simulation using these benchmark tests was performed in two stages: following the first series of tests, a preliminary analysis of the results was made and, on the basis of this, modifications to the PLL system were made before further tests were run. The remainder of this chapter describes the first series of test and the analysis of their results. Chapter 9 is devoted to a description of modifications made to the system in the light of the initial test results, and the subsequent tests.

8.5. Initial Benchmark Testing

8.5.1. Introduction

The first set of tests performed using the family database benchmark was intended to explore the effect of varying the number of processing elements and busses in the machine, the process scheduling algorithm remaining constant throughout this phase of testing.

Measurements were made of the time taken to run each query to completion under a range of different processing element and bus configurations. The times shown below represent the time taken to complete the evaluation of the queries:
aunt(x y)?
firstcousin(x y)?
sibling(x y)?
colour("red" w x y z)?
under different configurations (Fig.8.1). The data produced by the full range of queries is listed in Appendix G; performance figures for the other queries show a similar pattern to those listed below.

Two conclusions could be drawn immediately from these tests: first that increasing the number of processing elements led to a reduction of query response time in all cases, indicating that the overheads involved in the parallel system were small enough to allow good parallel speedups. Typical results were speedups in total evaluation time in the range of 39 - 28 for a system with 50 processing elements. (The base line for these comparisons has been taken to be the (Query evaluation time on 2PEs/1Bus machine)*2. It would seem more appropriate to use the actual figure for the sequential interpreter but as will be discussed in Chapter 9.6, this could lead to inaccuracies owing to the fact that the duration of certain data copying activities are discounted in the parallel system but not in the sequential one.

Query	Processing Element/Bus Configuration									
	100/5	100/2	100/1	50/5	50/2	50/1	20/5	20/2	20/1	2/1
aunt	48	50	48	60	62	62	100	101	103	843
first-cousin	488	488	481	551	641	559	1121	1182	1221	9280
sibling	34	34	34	44	44	44	84	84	84	631
colour	85	85	82	137	140	136	297	297	296	2707

Fig. 8.1 - Total Query Evaluation Times in ms (Initial Test Series)

The comparative performance of the sequential and parallel systems are shown in more detail in Fig.9.11.

These figures indicate that there was a significant amount of OR parallelism within the test program and that the overheads involved in exploiting it were small enough to allow significant performance gains to be achieved. The exact amount of these overheads will be discussed in the light of the next conclusion.

The second inference to be drawn from the figures was that the number of busses used per number of processing elements made almost no difference to the response time for these type of queries. In general queries were answered as quickly on a single bus system as a multiple bus one. The search for an explanation of this forms the basis of this preliminary analysis. In order to assess the relative usage of the bus network in comparison with the processing elements it is necessary to look at the patterns of processing and message passing.

8.5.2. Processing and Data Transmission Times

The following set of data (Fig.8.2) gives details on the processes involved in the query;
aunt(x y)?
The full set of data for this query is available in Appendix G and the following table gives a summary of the data. The timings given are average process times for the each type of process during the run and were obtained using a 50 processing element/2 bus configuration (although the number of processing elements and busses do not affect the processing times of individual processes). "Set-up", "rewrite" and "spawn" times are defined in Chapter 8.3.2 and refer to the three stages that occur in a process; non spawning processes represent the leaf nodes in the solution tree, and will either result in failure or a binding set.

Processes	No.of Procs	Set-up Time/Proc	Rewrite Time/Proc	Spawn Time/Proc	Words/ Data Pckt	Transfer Time/Pckt
Spawning Processes	33	140	8465	3327	8.8	12.2
Non Spawn. Processes	678	129	1732	---	---	---
Total	711	129	2044	---	---	---

Times in microsecs

Fig. 8.2 - Average Process Timings with Query aunt(x y)?

These figures show clearly that for this query the average packet transfer time was tiny in comparison with the process times. When processing times were considered it was found that non spawning processes are in general shorter in their rewrite times than spawning processes because so many represent FALSE returns where conflict in binding values leads to failure early on in the rewrite process. In this particular query, of the 678 non spawning processes, 670 responded FALSE and 8 resulted in valid variable bindings.

Thus the situation exists where broadcast communication of data from one processing element to several others is very efficient, because of the speed of the bus and the compaction of data into an optimised packet but processing of individual processes is slow in comparison.

In this particular query, because the data packet sizes were small, the transfer times were negligible in comparison with process times. However in a different type of query where, for example, a large list of bindings was regularly passed in the data packet, the result would be a sizeable data packet and hence a longer transfer time. However even if average packet size were increased by a factor of ten, the imbalance between processing and message passing would still be very great. It is necessary therefore to look in detail at the evaluation of processes to attempt to pinpoint any area in which inefficiencies exist.

The apparent inefficiency in process execution may arise from three sources; first overheads in the software due to the requirements of the simulation, secondly from the operation of spawning processes, or thirdly in the actual rewriting of the expression tree. The fourth possibility, namely lengthy processing to "set up" the process, ie to re-establish the expression tree, appears from the figures to involve a small proportion of processing time and is thus not considered at this stage.

8.5.3. Simulation Overheads

The process of modelling a parallel system in a single machine has been achieved by the multiple use of the same stack area for different processes. This produces the need to create multiple copies of data structures in order to ensure that independent processes are not corrupted by earlier operations in the same area of the stack. This has been discussed in Chapter 5.4.4.5 in the section describing the implementation of the parallel interpreter, and because it was recognised that the copying of data structures representing processes would not be needed in a parallel machine with no local memory, it was deliberately decided to exclude the time taken in performing this from the times measured. Thus the major overhead due to the simulation requirements has already been discounted.

There are some minor overheads that can be pinpointed and these have not been excluded from the measured processing figures. They include extra parameter passing to allow control information on the processing element involved to be passed from one rewrite function to another, as there is only one copy of the rewrite rules in the simulation. In a "real" machine each processing element would hold its own copy and the extra parameter is not needed. However this will only be responsible for a small increase in processing time and in terms of explanation of long processing times the simulation overheads cannot provide the answer.

8.5.4. Spawning Overheads

The amount of total process execution time spent in spawning of processes is small when query evaluation as a whole is looked at. This is largely due to the fact that the majority of processes are non spawning, certainly for the types of rule base which give potential for good OR parallel execution. Within the spawning processes, the proportion of the total time spent on the spawning operation is typically in the 25-30% region for the queries put to the family database.

As the software is written at present process spawning can be divided into four tasks as described in Chapter 5.4.4. To recap, these are:
a) when an OR node is encountered in the rewriting of an expression, new process structures are created to represent each OR node branch; these "processes" hold control data and the pointer to the appropriate branch of the OR node; this pointer forms the first element in the process description component of the process; the processes are chained together on a temporary queue within the appropriate processing element;
b) if an OR node has been encountered within the rewriting of an AND node, the system walks back up the AND-tree marking it with a special node to indicate that OR processes have been created;
c) if the rewriting of an AND node has involved its marking as described in b), a further recursive operation now retraces the AND tree creating a list of conjoined nodes, the process halting when no further nested ANDs are found; a pointer to this list is attached to the end of each process description in the new process structures;

d) a list of binding values is formed and added to the end of the each process description list in the process structures; the stack locations and the shared binding list are then reset.

Two points emerge from this description of process spawning: first in certain aspects the code is inefficiently organised, particularly in its almost exclusive reliance on recursive functions, and the repetition of the same tree walking involved in b) and c). These inefficiencies could be eliminated by recoding the operation using better algorithms and an iterative approach.

However the other point involves the fact that the intended organisation of data in the processing elements of the parallel machine would render unnecessary some of the processing described above. At present the interpreter creates separate process structures to represent every individual spawned process; however when the system is implemented on a broadcast architecture there is no intention to form separate data packets for each process - rather a composite packet is to be broadcast to a number of processing elements. Thus the processing that is currently performed in phase a) will be reduced to a single operation, namely the marking of the parent node of the OR tree, which is to form the first pointer in the broadcast data packet. The need to create a separate list of bindings values and to clear the stack each time a process finishes is a response to the simulation situation where the stack is shared between processes but need to be modelled as being local to each process. This operation is not needed in the real machine. Thus with the near elimination of processing time spent on a) and d) and the recoding of phases b) and c), it is likely that the spawning operation can be reduced significantly. The probable extent to which this can be done has not been quantified because analysis of the overall behaviour of the system has indicated that if the performance is to be significantly improved, the main area of concern must be the rewriting of expression trees. Even if process spawning could be reduced to a negligible operation the pattern of evaluation would not be materially effected. The reasons for this are considered in the next section.

8.5.5. Rewriting Overheads

The main time spent within a process whether a spawning or non spawning one is involved in rewriting, ie the basic PLL interpreter operations. In the queries in the family database the percentage of total process time spent in the rewrite operations stretched from over 99% to the region of 65%. Because the majority of processes are non spawning ones the total time spent in rewrite operations as a fraction of the overall processing time is very large, and hence it is to this area that attention must be focused if substantial improvements in processing time are to be achieved.

It is recognised that the modification of some of the node evaluation functions to accommodate the parallel simulation must have introduced some processing overheads. In order to check that the lengthy rewriting times were not occurring as a result of these additions, several tests were run using the original sequential and the new parallel versions of the PLL interpreter. For these tests, queries were put to a rule base which contained rules defined in terms of lengthy conjoined expressions but no disjunctions. This meant that the parallel system did not spawn processes but evaluated each query in a manner directly comparable to the sequential one. Details of the queries and the rule base are given in Appendix H, the following table (Fig.8.3) summarises the results.

Query	Parallel PLL			Sequential PLL
	Setup Time	Rewrite Time	Total	Total
Query0	35	5044	5079	4879
Query1	34	6947	6981	6901
Query2	34	4088	4122	3954
Query3	34	4032	4066	3848
Query4	34	827	864	836

Times in microsecs

Fig. 8.3 - AND Query Evaluation Times

The figures show that the overheads in the parallel system where no process spawning is involved are small - for these queries in region of 2% for the actual rewrite times.

The conclusion has to be made that the time spent in the application of the rewrite rules accounts for the vast majority of processing time in the parallel system. The present design of the machine provides for performance benefits by virtue of the exploitation of OR parallelism in the system and the ability of a processing element to transfer data simultaneously to number of other processing elements, but provision of a multiple broadcast facility is under-used in this type of rule base.

8.6. Summary

This chapter has described the initial testing and results generating phase of the project. The original plan for testing the system has been described in Chapter 8.2 where the required results are discussed. The intention was that after the analysis of the overall performance and processing/communication overheads had been accomplished, the different scheduling methods and processor utilisation would be looked. However it had not been expected that the imbalance between overall processing and communication times would prove so great, and it was decided at this stage to look more carefully at the reasons for this. On the assumption that the calculated data packet sizes and transmission times are close approximations to those produced in the real machine, it is necessary to look at the PLL interpreter to see if performance increases can be achieved for it.

Although the spawning times in spawning processes are not insignificant at present, ways in which this aspect of the system can be speeded up considerably have been discussed (Chapter 8.5.4). The main area in which lengthy processing appears to be taking place is the core of the PLL system, ie the rewriting of rules and further analysis of its behaviour is called for. This is addressed in Chapter 9.

Chapter Nine

Tests and Results from the Modified Parallel PLL System

9.1. Introduction

This chapter addresses the concerns raised in Chapter 8 regarding the performance of the parallel interpreter. Further tests have been devised to measure aspects of its behaviour and the results are presented below. Fig.9.1 provides a diagrammatic description of the relationship of the various tests and includes Chapter and Section references for each series of tests.

9.2. The Performance of the Rewrite Interpreter

The initial simulation results have indicated that the execution of the interpreter code is primarily responsible for the apparent large discrepancy between communications and processing times. In order to achieve improvements in overall system performance, methods of handling the rewriting task more efficiently have to be considered. This is a major area for future investigation, and the results presented in the following sections are intended to provide an indication of the type of performance improvements that could be achieved by judicious recoding of the interpreter. A more radical approach would be to move towards a fully compiled system as used in most efficient Prolog implementations but this lies outside the scope of this project.

Inspection of the code for the present interpreter reveals a heavy reliance on the use of functions which individually perform small computational tasks, and in particular the control of execution by means of recursive function calls. A system which uses trees as data structures is likely to be implemented by means of recursive algorithms because they make the task of the programmer much easier. However there is an implementational price to pay for this conceptual simplicity, and the overheads of function calling are likely to be significant. It was decided to look in more detail at these overheads in relation to the interpreter code. By attempting some form of quantitative analysis of overheads within execution patterns of the present interpreter it was hoped to be able to predict the possible improvements that could be obtained from recoding the interpreter while maintaining the same specification of its operation.

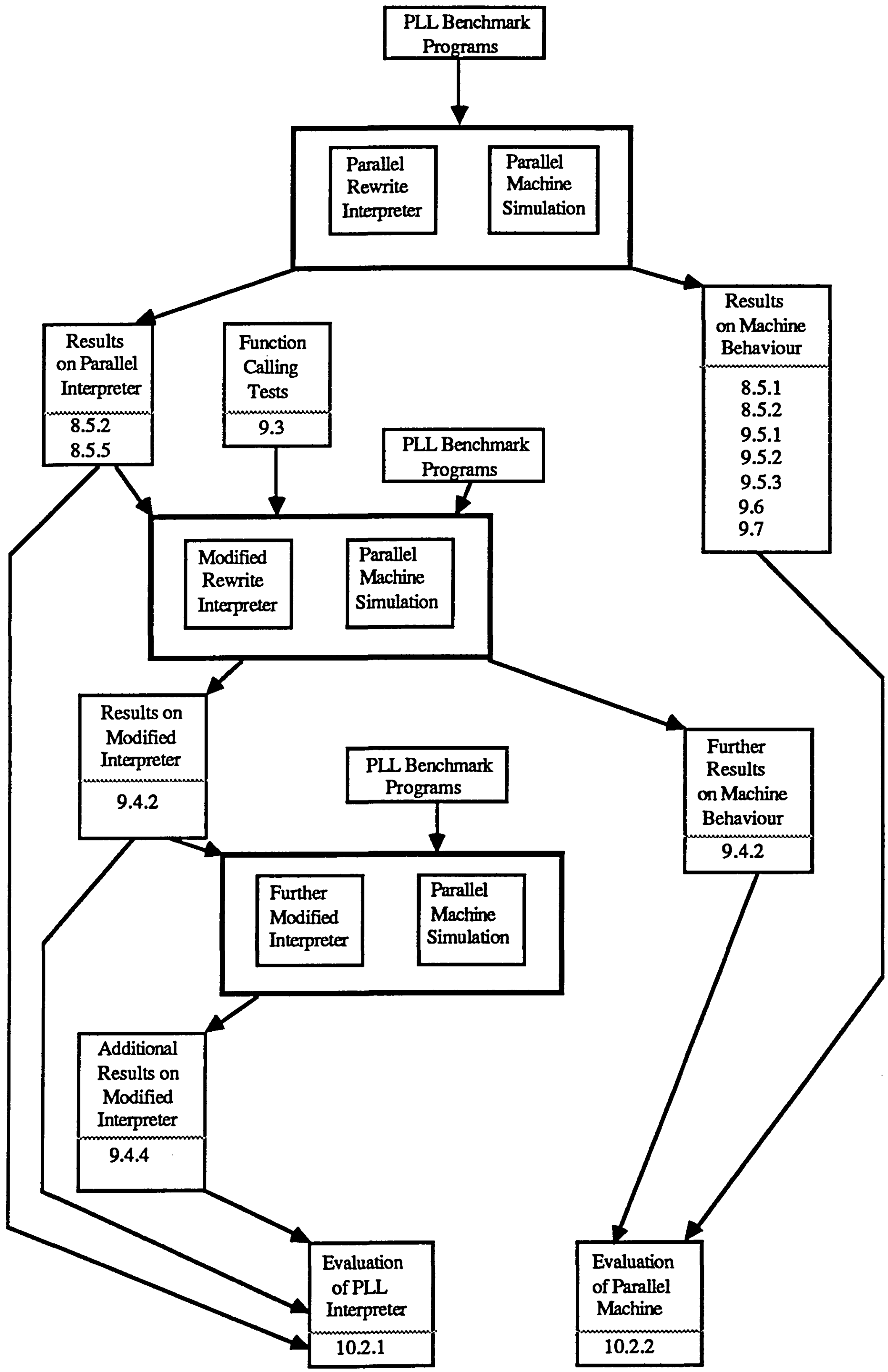


Fig. 9.1 - Testing Summary with Chapter/Section References

The first stage in determining the overheads in the interpreter code was to look at the general timings involved in C function calls using the Transputer based system. Comparative tests were also performed using the Sun 3/60 system although it was appreciated that the limitations in the granularity of timings would not allow for the incorporation of the information into the Parallel PLL system running on the Sun - see Chapter 7.4.3.3. The production of timing data from the Transputer system was made more complicated by the architecture of the Transputer which includes both "on chip" and external RAM. Access time for external RAM is considerably slower than that for internal RAM and care needs to be taken in the placing of program code and work space if timings are to give valid comparative results. In order to emulate the performance of larger programs this function testing program was configured to run on external RAM only. The details of the system configuration under the 3L Parallel C system are given in Appendix J.

9.3. Function Calling Overheads

9.3.1. Measurement of Function Calling

The following tests were performed to gain some indication of the performance overheads involved in function calling in the Transputer and Sun 3/60 systems. The first table (Fig.9.2) shows the time taken for 100,000 iterations of various loops. The contents of the loop ranged from a null operation, ie ";" in C, to a function call with a number of parameters. In each case the body of the function did no computational work: in the case of void functions the function performed the null operation, and in the case of functions returning an integer the body was merely a return of one of the parameters passed to the function. The full code for the test program is given in Appendix I.

The graph (Fig.9.3) shows the effect of increasing the number of formal parameters to a function. Two functions were used in this test: the first was a void function which did no computational work (Function 1), the other performed a simple arithmetic task involving one of the parameters and a local variable (Function 2). The results refer to measurements of the time for 100,000 iterations of each function: Function 1 was tested on the Sun 3/60, and both functions on the Transputer system. The time taken by the arithmetic operation included in Function 2 was separately measured at 350

Function	Sun 3/60	Transputer
No function - null operation	183	283
No function - assigment of variable	216	365
Void function - no parameters	416	635
Returning function - no parameters, no assignment	449	698
Returning function - no parameters, with assignment	466	766
Void function - 1 parameter	449	697
Returning function - 1 parameter, no assignment	583	828
Returning function - 1 parameter, with assignment	666	856
Returning function - 2 parameters, no assignment	583	856

Times in ms for 100,000 iterations

Fig. 9.2 - Timing of Functions

ms/100,000 iterations. Tests were also performed to measure the effect of the number of local variables within a function. It was found that the introduction of local variables per se had no effect on the timing of function execution. Only when the local variable was operated on, eg by being assigned a value, did the function time increase. Inspection of the assembler listing of the program produced by the "decode" utility showed that the compiler had ignored local variables declared but not used. For those local variables which were used in the function code space had been allocated at compile time. Thus it was concluded that the introduction of local variables was not imposing timing overheads on function calling times.

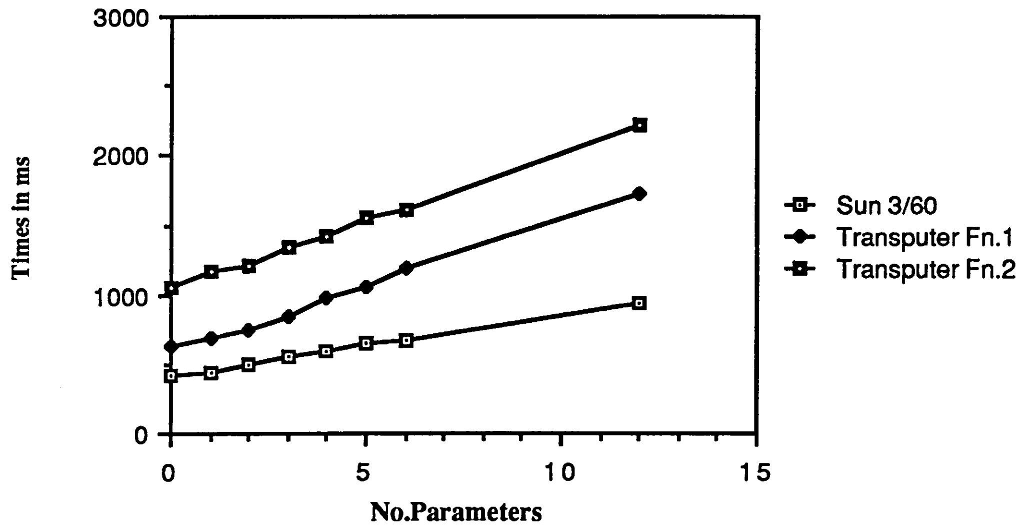


Fig. 9.3 - Effects of Parameters on Function Times

9.3.2. Allowance for Function Calling Overheads

The basic conclusion to be drawn from these small experiments were that function calling does cause performance degradation and this increases with the number of parameters involved. Recursive definitions although conceptually attractive impose constraints on performance when compiled into low level code.

The next step in the analysis of the performance of the PLL rule rewriting code should ideally be an attempt to recode some or all of it using iterative methods and an advanced compiler. However the amount of work involved in this meant that it was not possible to consider within the time scale of the project, and therefore a second approach had to be considered. This involved inspection of the code adding a performance overhead factor to each function and arranging for a running total of these factors to be maintained during evaluation of a process. The new process times would be calculated from the actual process times less the total processing overhead factor. Allowance had to be made also for the times involved in keeping a tally of the processing overhead factors.

The typical function in the parallel rewrite interpreter is a function returning an integer or pointer to an integer. The average number of formal parameters lies between 2 and 3 and these are almost invariably integers or pointers to integer values. Based on the figures from the table shown in Fig.9.2 it was decided to designate a function calling overhead of 6 microsecs which could be applied to each function. At this stage tests were performed to time the overhead in incrementing a running total which would have to be maintained throughout each process. This was identified as being in the region of 1 microsec, and this was added to the function overhead figure, giving a final value of 7 microsecs as the "cost" of performing a function in the PLL system. It was recognised that this was a fairly crude measure of the overheads of function calling in the execution of queries to the system, but it was intended to be used as an indicator to the performance degradation attributable to this source, not an exact quantitative measure.

9.4. Results of Revised Tests

9.4.1. Introduction

The same benchmark tests used for the initial testing were applied to the second version of the interpreter. As described in the previous section a function calling overhead factor of 7 microsecs was introduced into each function and a running total kept throughout the execution of each process. This total was subtracted from the measured process time and this revised time was used to represent the predicted "optimised" process time. As before subtotals for "set-up", "rewrite" and "spawn" time were maintained in order to give comparative data for analysis of the benefits to be obtained by reducing function calling.

9.4.2. Revised Performance of the Interpreter

The results given in Fig.9.4 show the timings for the query "aunt(x y)?" summarised in the same format as the original test results (see Fig.8.2). Full results are available in Appendix G.

The results show that when allowance is made for function calling overheads, the predicted performance of the interpreter improves. The average number of function calls in a process bears a fairly close relationship to the length of time a process takes: for each 100 microsecs of

Processes	Set-up Time/Proc	Rewrite Time/Proc	Spawn Time/Proc	Total	Functions/ Proc	Functions/ 100 microsecs
Spawning Processes	115	6584	2509	9208	671	~7
Non Spawn. Processes	109	1409	---	1518	~90	~6

Times in microsecs

Fig. 9.4 - Average "Optimised" Process Timings for Query aunt(x y)?

process time approximately 6 - 7 function calls are involved. If this is related to the original estimate for function overheads, ie that each function call adds 6 microsecs to the processing time, the elimination of function calls should result in processing times being cut by 40%. The data in Fig 9.5 shows the results for four queries when the "optimised" version of the parallel PLL interpreter was used.

Query	Processing Element/Bus Configuration								
	100/5	100/2	100/1	50/5	50/2	50/1	20/5	20/2	20/1
aunt	39 *48*	38 *50*	38 *48*	45 *60*	46 *62*	51 *62*	80 *100*	78 *101*	78 *103*
first-cousin	388 *488*	435 *488*	386 *481*	428 *551*	507 *641*	460 *559*	917 *1121*	979 *1182*	824 *1221*
sibling	27 *34*	27 *34*	27 *34*	35 *44*	35 *44*	35 *44*	69 *84*	69 *84*	69 *84*
colour	68 *85*	69 *85*	68 *82*	109 *137*	110 *140*	116 *136*	232 *297*	237 *297*	237 *296*

Times in ms

Figures in Italics refer to Initial Test Series Measurements

Fig. 9.5 - Total Query Evaluation Times for "Optimised" Version

The values in Fig.9.5 show a general improvement in performance somewhat less than the anticipated 40%. Further investigation of this indicated that the allowance of 1 microsec for updating the function call count was probably an underestimate as the manner in which it was implemented involved keeping a running total of two variables. However the improvement in performance due to removal of function calls is not likely to exceed 40%. It has been suggested that time spent in the spawning phase can be reduced substantially by the move to a composite data packet as opposed to a number of process structures. Even if it is assumed that this aspect of process evaluation can be reduced to a negligible figure corresponding to set-up time, the above results show that it is questionable as to whether the elimination of function calls from the rewrite phase will provide enough scope for future performance optimisations on which to base a realistic implementation.

9.4.3. Implications for Further Testing

In view of the conclusion that sufficient performance improvement may not be achieved by merely removing excessive function calls, the usefulness of other test results is called into play. As results on the pattern of return of results and input memory usage had already been obtained for the original and "optimised" code test runs, it was decided to evaluate these and this is discussed in Chapter 9.5.1 and 9.5.2. This analysis is valid under the present method of implementing rewrite rules.

Although it was appreciated that detailed proposals for the design of an improved interpreter lay outside the scope of this project it was felt important to obtain more information about the pattern of function calling within the present interpreter. The results produced by the "optimised" version of the interpreter merely gave the total number of function calls for each process and it was decided to augment this with details about the actual functions involved. The results shown previously (Fig.9.4) reveal that large numbers of function calls are involved in each process evaluation, and a series of new tests were devised to obtain a more accurate picture on the role of these functions.

9.4.4. Details of Function Calls in the Rewrite Interpreter

In order to obtain a clear picture of the pattern of function calls in a typical query response, the rewrite interpreter was amended to keep a running total of calls to each function during the evaluation of a process. These were written to file at the end of each process thus giving information on the type and number of function calls involved in the process.

More than seventy separate functions could be used during process evaluation: for analysis purposes these were grouped into eight categories. These were:

Category 1: top level evaluation functions,
eg <eval_andP>, <eval_notP>, <eval_plusP>.
Category 2: lower level list and expression evaluation functions,
eg <eval_list_to_valP>.
Category 3: lower level rule rewriting and expansion functions,
eg <expand_ruleP>, <expand_rule_listP>.
Category 4: lower level arithmetic functions,
eg <node_plusP>, <node_multiplyP>.
Category 5: variable installation and instantiation functions,
eg <init_varP>, <set_varP>.
Category 6: process structure creation functions incl. spawning functions,
eg <spawn_or_processes>, <create_process>,
<create_process_desc>.
Category 7: memory space creation functions,
eg <node3>, <node2>, <copy_exp>,<copy_list>.
Category 8: garbage collection functions,
eg <release_node3>, <release_process>, <release_exp>.

In Appendix G results from two queries:
aunt(x y)?
and
stepparent(x y)?
are given and demonstrate the numbers of function calls in each process, decomposed into the eight different categories. Fig.9.6 reproduces two examples of these results taken from the query:
aunt(x y)?

The two processes referred to in Fig.9.6 represent a non spawning and a spawning process: although the pattern of function calls varies from process

Process	Function Category								
	1	2	3	4	5	6	7	8	Total
No.284 Spawning									
Set-up	–	–	–	–	–	4	2		6
Rewrite	181	6	163	–	2	–	243	29	624
Spawn	--	–	–	–	–	112	47	–	159
Total									789
No. 285 Non Spawn									
Set-up	–	–	–	–	–	4	2	–	6
Rewrite	23	6	–	–	6	–	19	43	97
Spawn	–	–	–	–	–	–	–	–	--
Total									103

Fig. 9.6 - Numbers of Function Calls within Processes

to process and query to query, they can be regarded as demonstrating certain typical features about function calling overheads. It can be seen that in the spawning process, the spawning component of the task involves a large number of calls to the functions which set up the spawned process structures. Because these are recursively defined each additional element to be added to a process structure involves a separate function call. The memory creation functions also play a significant role as each new "request" for stack space requires a function call. However as discussed in Chapter 8.5.4 this method of organising the spawning task would be substantially modified for implementation in the "real" system, so further consideration of this aspect is inappropriate.

The "set up" phase of both processes involves a small number of function calls and as the earlier timing data demonstrated, this is not contribution significantly to process timing. For both spawning and non spawning processes the "rewrite" phase accounts for the largest time slice and the greatest number of function calls during process evaluation.

When the function calling of the rewrite operation is looked at in more detail the groups of functions involving significant numbers of calls can be identified as:

a) top level evaluation functions (Category 1),
b) rule expansion functions (Category 3),
c) memory space creation functions (Category 7),
d) garbage collection functions (Category 8).

Function Group	Process Type	
	Spawning	Non Spawning
Top Level Eval	29	24
Rule Expansion	26	0
Memory Creation	39	20
Garbage Collection	5	44

Fig. 9.7 - Function Group Percentage during Rewrite Phase

The approximate percentage of the rewrite phase that each of these groups occupied in the two example processes is shown in Fig.9.7. For both processes top level evaluation functions play a significant role as do memory space creation functions. However no rule expansion is taking place in the leaf or non spawning process, whereas approximately one quarter of the function calls in the spawning process is concerned with this lower level user rule rewriting operation. This represents the copying of the right hand side of the rule onto the evaluation stack each time a user defined rule is applied. Because of the recursive nature of the rule expansion functions each element in the right hand side expression requires a separate function call to implement copying. When this is added to the memory space creation function calls which are also involved in making a copy of the "expanded" rule the overheads in this operation are significant.

In contrast garbage collection forms an important role in the function calling pattern in the non spawning process. This is a response to the manner in which the evaluation stack has been used in the parallel interpreter software (see Chapter 5.4.2). Instead of a global reset at the end of process evaluation, discriminatory space retrieval has to be performed. This overhead would be minimised in a move to a "real" multiprocessor system.

These results serve as an explanation for the large total number of function calls involved in process evaluation as reported in Chapter 9.4.2, and also indicate the areas where revision of the interpreter would be most effective.

9.5. Additional Tests on Machine Performance

9.5.1. Return of Results

The size of an individual results packet produced during the benchmark tests was typically small, and for programs that do not rely heavily on list structures this is likely to be the case. Unlike the data packets used to convey spawned process information, results packets do no contain intermediate introduced quantified variables: their size is normally determined by the number of variables which the user inserts at query time. For example, if the query "firstcousin(x y)?" of the family database, all results packets will take the format of binding values for x and y, giving a total packet size of four words (see Apendix G for detailed results). However the data packets used to spawn processes during the evaluation of this query may contain up to fifteen words. It is generally true therefore to assume that results packages will not be excessively large and given that the performance of the present system is apparently determined by processing not communication times, delays due to irregular pattern of results return are likely to be correspondingly insignificant. Only when the system is modified to decrease the time spent in rewriting individual processes will the overhead in return of results need to be considered in detail. The present hardware design which assumes a single bus to carry results data packets back to the controller and thence to the user works perfectly adequately because of the imbalance between processing and communication.

9.5.2. Input Memory Usage

Values were obtained for the maximum storage requirement for each input memory in every processing element during query evaluation. These were recorded for the original and optimised versions of the interpreter as it was not clear what effect that improved processing speeds would have on the input memories.

Because each bus is directly connected to an individual input memory in a processing element the amount of data in each input memories depends on which bus has been used for the transmission as well as which processing element has been designated receiver for the process, ie bus scheduling and load balancing between processing elements both contribute to the pattern of usage of input memories. However these scheduling mechanisms are independent of each other and Chapter 6.5.3 has discussed the possibility that this could lead to wide discrepancies in the use of these memories. It was hoped that simulation results would give an indication of whether this is likely to happen. Although the results are still subject to the same proviso that they do not reflect the necessary "ideal" system because of processing/communication imbalance, it is nonetheless of interest to look generally at the type of input memory usage obtained in the two sets of tests. The table (Fig.9.8) shows the memory usage for the query "firstcousin(x y)?" in the original version and the one which allows for function calling overheads. This is designated as the "optimised" code. The table refers to the situation where the query is executed with fifty and twenty processing elements and five and two busses respectively. The detailed figures showing individual values for input memories in every processing element are given in Appendix G.

Machine with 2 Bus Configuration				
Input Memory Number	50 PEs		20 PEs	
	Original Code	Optimised Code	Original Code	Optimised Code
Memory 0	56 - 121	65 - 123	126 - 167	139 - 249
Memory 1	42 - 123	54 - 123	126 - 167	150 - 237

Machine with 5 Bus Configuration				
Input Memory Number	50 PEs		20 PEs	
	Original Code	Optimised Code	Original Code	Optimised Code
Memory 0	13 - 67	13 - 66	56 - 98	56 - 122
Memory 1	15 - 56	15 - 53	43 - 109	54 - 109
Memory 2	15 - 57	28 - 67	42 - 96	81 - 111
Memory 3	13 - 68	26 - 83	56 - 96	83 - 123
Memory 4	13 - 56	13 - 83	57 - 86	56 - 111

Memory Size in Words

Fig. 9.8 - Maximum Input Memory Usage with Query firstcousin(x y)?

Two implications can be drawn from these tests: the maximum size required for input memories related to a given bus does vary from processing element to processing element, a spread of 56 to 98 representing a typical variation. However when viewed over all the input memories within a given processing element there is no marked imbalance in favour of a particular memory. This would indicate that the round robin approach to scheduling busses does result in a reasonable distribution of data in the input memories.

The second point revealed in the figures is that when the "optimised" code is used, the overall storage requirement for input memories does not differ substantially from the original version. Some of the figures would appear to indicate an increased need for buffer space for incoming processes in the optimised version but the statistical significance of these observations has not been determined.

9.5.3. Load Balancing Strategies

The previous tests relating to the original interpreter and the "optimised" version have used the load balancing mechanism which allocated processes to processing elements depending on the "busy-ness" of each processing element. This "busy-ness" measure has been determined by a count of the number of processes awaiting execution in each processing element. This seemed a reasonable first approach to take and it was intended to explore other possibilities for allocation of work to the processing elements. However in view of the imbalance between processing and communication, it was decided that there was little point in obtaining measures of efficiency of scheduling at this stage. The future efficiency of the system lies with methods of improving overall processing speed and only when this is achieved can load balancing be looked at realistically.

Before looking at some of the theoretical considerations involved in load balancing it was decided to check that the present scheduling algorithm was producing some performance benefit, and a limited number of tests were performed using the original interpreter code but allocating processes to processing elements on a purely random basis. The results are summarised below in Fig.9.9 and show that for all queries the random load balancing policy led to slower overall performance. It is therefore safe to assume that the decision to allocate processes on the basis of "busy-ness" of processing elements is providing substantial benefits.

However the scheduling policy as implemented at present is not providing maximum performance benefit. This can be seen when the chart of processing element usage is considered for the query "stepparent(x y)?":

Fig.9.10 shows a schematic repesentation of the usage pattern produced by various test runs for this query. In the situation where there are considerably more processes for execution than processing elements, an ideal load balancing mechanism would ensure that for the majority of query

Query	Processing Element/Bus Configuration			
	50/5	50/2	20/5	20/2
aunt	81 *60*	82 *62*	124 *100*	126 *101*
first cousin	917 *551*	818 *641*	1642 *1121*	1448 *1182*
sibling	64 *44*	64 *44*	95 *84*	95 *84*

Figures in Italics refer to Results from Initial Series of Tests

Fig. 9.9 - Total Evaluation Times in ms with Random Scheduling

evaluation time, the maximum number of processing elements were in use. As can be seen in Fig.9.10 and Appendix G this is not the case with the examples given.

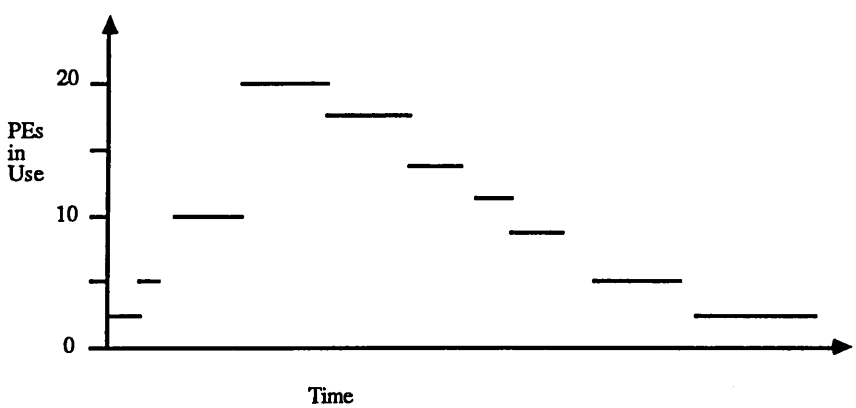


Fig. 9.10 - Schematic Representation of Processor Usage

The difficulty with load balancing is in predicting how long a given process is going to need to complete its execution. This has been discussed in the section on the granularity of processing (Chapter 3.2.2) and it can now be clearly seen that the process based parallel model for the PLL produces a mixed granularity system. In the response to the above query "stepparent(x y)?", the execution times for individual processes ranged from 1564 to 33,422 microsecs. The majority of non spawning, ie terminal processes, were comparatively short lived, reflecting the fact that the AND node rewriting rule produced the result FALSE at an early stage. However other non spawning processes failed at a much more advanced stage in the AND rewriting, giving longer execution times, and four processes went on to produce binding values for x and y which also involved lengthy evaluation times.

It would appear that there is no independent measurement that can be made before the start of execution to determine length of time a process is likely to take to complete execution. The size of the process description, ie the number of words in the data packet which inaugurates the process, appears to bear no relationship to the eventual processing time: in the query used for these tests, ie "stepparent(x y)?" the data packets produced were all either eight or nine words in length. Chassin de Kergommeaux discusses this problem in relation to the ECRC PEPSys system and concludes that it is important to minimise the number of short lived processes created, because of overheads in process creation and load balancing considerations [Chassin de Kergommeaux 89]. For the parallel PLL the overheads in process creation and spawning are relatively low but the problem of load balancing in a mixed granularity system remains, and it is difficult to see how this can be ameliorated given the current PLL method of handling rewriting.

The load balancing problem was demonstrated when repeated measurements were made for the same query running under identical machine configurations. As shown in Appendix G there was a variation in the overall query evaluation times produced by different runs. These were caused by slight differences in the measured, ie "real", time of the process resulting in different allocation patterns for each run, and thus to different query response times. For queries with wide variations in the length of individual processes this may lead to noticeable differences in overall response times. For the purposes of the examples given in this chapter the shortest query response time obtained has been used as the stated measurement.

9.6. Performance Benefit due to Parallel Execution

The previous section has looked at the performance measurements of various components of the proposed architecture. It is relevant at this stage to consider the system at a macro level and attempt to quantify the overall performance benefit to be gained by the introduction of OR parallel execution.

As has been shown the ratio of communication to processing times is so small that for the purposes of this analysis communication times can be discounted. In a more realistic system it is recognised that this is not likely to be the case particularly when the transfer of data from disk to the processing elements is involved. However the present discounting of communication times means that any performance benefit during the execution of a query can be directly related to the amount of parallel execution taking place.

Measurements for total query evaluation time with varying number of processing elements for a number of different queries are shown below in Fig.9.11. These results were obtained from the original parallel interpreter and refer to a one bus machine configuration. When these were compared with the total execution times produced by the sequential interpreter it was found that the parallel interpreter configured with two processing elements and one bus evaluated queries in less than half the time taken by the sequential version. The explanation for this lies in the copying/spawning mechanism in the two systems. In the sequential system when an OR node is encountered multiple copies of the expression tree are produced, whereas the parallel interpreter produces multiple process structures. The operation of installing these process structures is less than the full scale copying that takes place in the sequential interpreter and thus the copying of OR expressions produces a larger time overhead than the corresponding process creation operation. To test the extent of this overhead a second series of tests were performed in the serial system to time its overall query response times excluding the time taken to copy expression trees when OR nodes were encountered. These results are displayed in the right hand column under the sequential interpreter results (Discounted Sequential Times). The calculations of speedup due to parallel execution are based on these figures: Fig.9.12 shows the speedups for each query in graphical format.

Query	Parallel PLL Simulation No. of Processing Elements						Sequential PLL	
	100	50	20	5	3	2	Original	Discounted
aunt	48	62	103	340	557	843	1773	1590
sibling	34	44	84	265	430	631	1255	1174
colour	82	136	296	1091	1811	2707	5951	4663
factorial	86	86	86	86	97	151	353	255

Times in ms

Fig. 9.11 - Comparison of Parallel and Sequential Evaluation Times

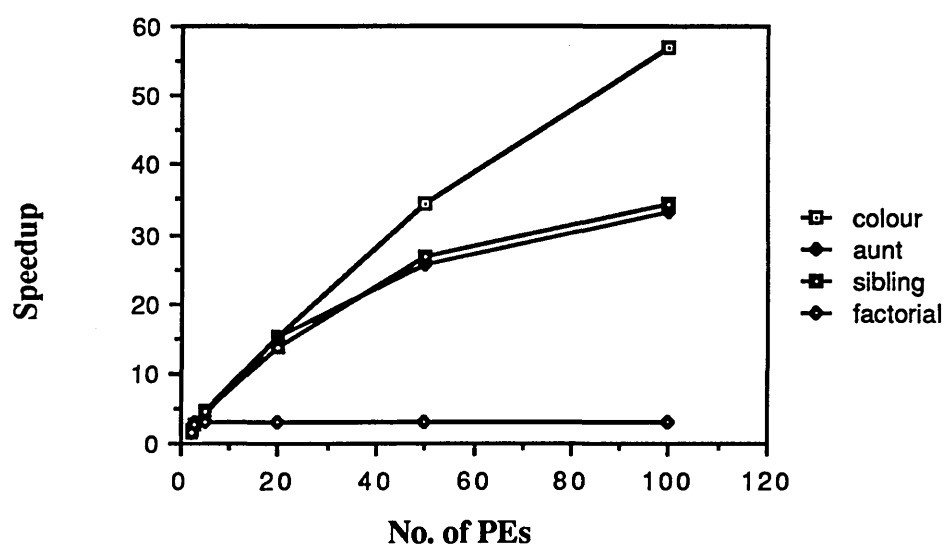


Fig. 9.12 - Graph of Performance Speedups

The comparative results indicate that a considerable amount of parallel execution is taking place during the evaluation of all the queries with the exception of "factorial(10 x)?". This confirms the original belief that for the Datalog type of program the introduction of OR parallelism is likely to prove beneficial. The pattern of process spawning involved in a query such as "factorial(10 x)?" does not result in a substantial number of candidates for concurrent execution.

9.7. Communication Delays

In the tests performed up to this stage contention for the communication medium occurred very infrequently, and in general the overheads involved in the whole operation of data packet transfer were small in relation to processing (see Appendix G for detailed results). However it is anticipated that alteration in the design of the interpreter will lead to faster rewrite operations, and therefore the time occupied in passing data round the machine will play a more significant role in overall performance. In order to give an indication of the effect of increased communication overheads, some sample tests were run in which the time taken for each process evaluation was artifically reduced by a factor of

Query	PE/Bus Configuration					
	100/5	100/2	100/1	20/5	20/2	20/1
colour	1210 *850*	1230 *850*	1746 *820*	2957 *2970*	3027 *2970*	2834 *2960*
aunt	524 *500*	554 *500*	659 *480*	1091 *1000*	1010 *1010*	1079 *1030*
sibling	409 *340*	446 *340*	539 *340*	848 *840*	845 *840*	843 *840*

Times in microsecs

Figures in Italics refer to Original System Times / 100

Fig. 9.13 - Total Query Evaluation Times for "Scaled" System

one hundred. This is clearly an exaggerated figure for potential performance improvement in the interpreter: to set it in context, some compiled Prolog systems developers have claimed performance improvements of up to thirty times when compared with their interpreted versions [IF Prolog 88]. The measurements listed in Fig.9.13 show the results of the tests on communication times. The figures in italic refer to the corresponding times for the original interpreter (see Fig.8.1); these have been scaled down by a factor of 100 to make comparison of the two systems easy.

It can be seen that for the queries run with the 20 processing elements configurations the number of busses made little difference to the total query evaluation times and they were very close to the times from the original version when appropriately scaled to match the speeded up interpreter. However for the 100 processing element machine there was a noticeable drop in performance as the number of busses decreased. The assumption is that in the smaller version all the processing elements have enough work allocated to them, and delays in receiving further data packets do not affect their "busy-ness". For the machine with a large number of processing elements the delay in spreading work around the machine becomes more significant as processing elements are "waiting" for work.

9.8. Summary

This chapter concludes the testing of the parallel system simulation. The performance of the rule rewrite interpreter has been analysed in detail in order to provide a basis for future work on its design. Further results on aspects of the performance of the parallel architecture have been presented. This information, together with the work discussed in Chapter 8, forms the basis for the evaluation of the system contained in the next Chapter.

Chapter Ten

Evaluation of the Project

10.1. Introduction

The aim of this chapter is to consider the design of the Parallel Pure Logic Language system in the light of the results obtained during the testing stage and to offer a critical assessment of the work. In addition the techniques used during the project are evaluated, and suggestions for future work in this area are presented.

The design of the parallel PLL system involves two distinct components: the abstract computational model for a parallel process based language system with its associated interpreter, and the proposed multiprocessor architecture. Although the two aspects have been developed concurrently they are not bound exclusively to each other: the parallel PLL language system could be mapped onto a different form of architecture, and similarly the bus based broadcast multiprocessor machine offers design features that could prove attractive to other applications [Brown 89]. The first part of this chapter looks at the design of the parallel PLL interpreter and then at the performance of the parallel machine in relation to the PLL. The second major chapter section contains a discussion of the manner in which the project was organised and the methods and tools used.

10.2. The Parallel Pure Logic Language System Design

10.2.1. The Parallel Rewrite Interpreter

The progress from a sequential logic language system based on rule rewriting to an OR parallel process based model has been documented in Chapters 4 and 5. The basic philosophy which prompted the work by ICL on the original system, ie the execution of pure logic, has been maintained in the move to a parallel system. Evaluation of the parallel interpreter can therefore be considered in two parts: assessment of the original sequential system and the degree to which the move to a parallel version has been successful. As the second aspect has formed a major part of the project this section will focus on it. However it is relevant here to look briefly at the original PLL system in the light of the work done during this project.

The first point to be made about the sequential system is that it is a research vehicle, not a commercial product, and the version that was used during the project was a comparatively early one. Subsequent versions have introduced more facilities and optimisations, although the basic mechanism of rule rewriting has remained constant [McBrien 88b], [Babb 89a], [Babb 89b]. The original inspection of the method of executing the language showed that many of the data structures involved in the organisation of the interpreter were similar to those used in current Prolog implementations (see Chapter 4.7). Search time during query evaluation was substantially reduced in comparison with Prolog by the method of "precompiling" the links between the user defined rules, and this would indicate that reasonably efficient performance could be expected with respect to interpreted Prolog systems. Work at ICL suggests that this is the case although no comparative measurement between PLL and Prolog execution speeds have been made during the course of this project [McBrien 88c]. However what has emerged from the testing phase of this project is that the method of implementing the handling of conjoined expressions can lead to excessively lengthy processing times. This has been described in Chapter 4.5.2. The rewriting of conjoined expressions was implemented in this manner in order to eliminate the order sensitivity problem which Prolog displays but it can impose a considerable and unpredictable performance penalty. As has been shown in Chapter 4.7 it also makes the move towards a fully compiled version of the PLL much more difficult.

The first decision that was made in relation to the design of a parallel version of the PLL was that the "purity" of the language would be maintained and that parallel execution would be the responsibility of the system and transparent to the programmer. It was recognised that this would have implications for two aspects of the design: first, if AND parallelism were introduced, some form of variable dependency analysis mechanism would be needed to allow shared variables to be recognised, and secondly the lack of programmer control over parallel execution could result in inefficiencies. Consideration of the type of programs used in knowledge based systems led to the decision to omit any AND parallelism. The implications of this decision are considered below.

Many parallel logic language systems have incorporated programmer control of parallel execution not only to circumvent the shared variable problem, but in order to ensure that parallel execution only occurs when it is likely to give performance benefits. In this way the programmer can utilise his or her knowledge of the program's behaviour and the target architecture to "fine tune" the performance of the system. The alternative position is that definition and allocation of parallel processing is the responsibility of the underlying system and the system has to incorporate mechanisms to ensure that these tasks are done as efficiently as possible. The discussion in Chapter 9.6.3 regarding load balancing in the parallel PLL system has shown that this is not a straightforward task. Because of wide discrepancies in the execution times of individual processes it has proved difficult to implement a good automatic scheduling mechanism. This will be looked at again in the assessment of the proposed architecture but it is recognised at this stage that the project has not been able to tackle this area in a particularly satisfactory manner.

The concentration on OR parallel execution at this stage has been discussed in Chapter 5.2 and 5.3. This was based on analysis of the type of programs used in the applications area under consideration, ie Datalog programs, and the experience of other research projects also points to the performance benefits to be gained from this approach. OR parallelism has the advantage that OR processes can be defined in a manner which makes them independent from each other. As has been shown parent processes terminate after their offspring have been created, and problems of two way communication between processes are avoided.

The test programs used to obtain performance measurements for the parallel PLL system were of necessity small, but nonetheless revealed the potential for a considerable amount of parallel execution. Figures given in Chapter 9.7 show speedups in the region of 30 to 50 times in comparison with the sequential version for typical queries involved in the small test programs. This would indicate that for a realistically large system the inclusion of OR parallelism is likely to prove attractive if overheads in process creation and communication can be kept at a manageable level. The decision to concentrate on OR parallelism appears to be vindicated by these results.

It is at this stage that the proposed parallel PLL system shows its individuality. Conceptually independent processes are spawned when alternatives are encountered during the course of query evaluation and these processes become candidates for parallel execution. The important feature about this spawning mechanism is that it is proposed to implement it by means of a broadcast operation. The one to many relationship between parent and offspring processes is formalised by the creation of a single data packet which can be interpreted by each offspring process in a unique fashion. As far as can be ascertained no other OR parallel logic system handles the spawning of processes in this manner. This method of passing data between parent and offspring processes can be seen as an amalgamation of copying and recomputing data (see Chapter 3.1.3.1). The advantage of this approach is that the overheads for process spawning do not increase with the number of new processes to be created. (This is not strictly true for the particular hardware implementation proposed - communication times do include a factor that relates to the number of processes involved, but this does not impose the high overheads that would be involved if each process was individually represented as a separate data packet. Conceptually the overheads of process spawning are linked to the amount of data contained in the combined data packet and independent of the number of processes involved).

The setting up of totally independent spawned processes can involve repeated, ie redundant processing. In the situation where the query:
(a(x) or b(x)) and c(x)?
is put to the system the two processes formed, ie
a(x) and c(x),
b(x) and c(x),
will both evaluate c(x). This aspect has been looked at in Chapter 5.3.2, and because of the different environments pertaining to the two evaluations of c(x), in general the computation is likely to produce different results, ie the computations resulting from the separate evaluations of c(x) are not identical and thus neither is redundant. It therefore appears reasonable to use this method of defining independent processes with the proviso that at particular stages in the computation a certain amount of repeated computation may take place. No attempt has been made to quantify the amount of repeated computation due to the method of process definition. It is known that repeated computation is involved in the standard rewriting of conjoined expressions and this is an area for further study.

The inclusion of a combined data packet which communicates information from parent process to its offspring by means of a broadcast operation appears to offer considerable scope for the mapping of the parallel PLL interpreter to a non shared memory multiprocessor machine, but before looking at the architectural and mapping issues further judgment is necessary on the present state of the parallel interpreter.

As has been shown there are aspects of the rule rewriting manager's operation that are inefficiently coded at present. This applies equally to the sequential and parallel versions. The heavy reliance on large number of small functions adds significant overheads to the performance of rule rewriting; this aspect has been discussed and quantified in the second phase of testing (see Chapter 9.4). The code which implements the spawning activity of the parallel version is equally inefficient, and no attempt has been made to optimise this. In Chapter 6.5.3 the proposed method by which the data packet is to handled within an individual processing element is discussed. It is likely that this approach which uses both the appropriate input memory and the output memory will reduce process initiation and spawning time, although it does not affect the rewriting phase of the process. Because of the uncertainty regarding the best method of improving the whole task of process execution it is difficult to make realistic predictions about the possible overall improvements which could be made in the future. Careful consideration has to be given to the advisability of attempting to optimise the interpreter using the same high level system rewriting rules; in the long run it may be more beneficial to take a more radical approach to improving performance in query evaluation.

10.2.2. The Bus Based Multiprocessor Architecture

The functional requirements for the proposed multiprocessor architecture were derived from the study of potential OR parallelism within the PLL and other Prolog systems. In Chapter 6.5 a possible realisation of these requirements has been discussed and the design of a multiple broadcast bus based parallel machine has been presented. The next stage in the project was to produce a simulation of the machine with the parallel PLL system mapped onto it, and make measurements of the predicted performance of the logic language system and the machine hardware. This section looks at the design of the multiprocessor machine and its performance when the parallel PLL system is mapped onto it.

Before looking at the measurements for the predicted performance of the parallel machine it is important to stress that the timings relating to this aspect of the system's performance are calculated values and not "real" ones. All the timing measurements made for the creation and rewriting of processes were made by calls to the inbuilt system clock, and thus represent actual times taken to execute a given task; in contrast the timings of data transmission, processor and bus allocation have been based on calculations derived from knowledge of the design features of the machine. As such there is no independent confirmation as to the degree of accuracy that they possess. Inaccuracies could be introduced into these calculations either by misinterpretation of the design implications or by mistakes in the coding of the calculations. It is hoped that neither of these aspects is causing incorrect values to be produced but the final confirmation of this can only be given by the construction of a prototype machine. As will be seen the accuracy of the calculated data transmission aspects of the system does not appear to be critical because of the processing /communication ratio.

The first point to make about the performance of the simulated machine is that the time spent on communicating data between processing elements was minute in comparison with overall processing times. This result was somewhat unexpected, although not unwelcome, as it had been believed at the outset of the project that communication overheads could limit the usefulness of a non shared memory system. The imbalance between processing and communication as measured by the simulation was considerable and this means that if the performance of the interpreter is substantially improved, the communication overheads should be maintained at an acceptable level for the type of programs used in the benchmark tests. This has been demonstrated by the series of tests in which the performance of the rewrite interpreter was artificially "improved" by a factor of one hundred.

The type of programs used in the benchmark tests typically produced comparatively small data packets (under thirty words in length) because there were no long list structures included. Delays in obtaining a bus do increase as the number of busses is decreased, but these account for such a tiny proportion of the total query evaluation time that overall performance measurements show no significant degradation with a diminishing number of busses, indicating that for this type of program the inclusion a multiple bus system is not necessary. Whether this is generally true for other types of programs is undetermined.

Two aspects of the design of the machine are worth considering in terms of the required functionality: these are processor and bus allocation. In the description of the hardware in Chapter 6.5.4 no details were presented as to how these functions were to be implemented in hardware as it was realised that when the performance predictions for the system became available the hardware requirements would be more clearly seen.

The performance measurements have shown that bus allocation does not need to be a complicated procedure: on the assumption that more than one bus is needed (and this is questionable for the PLL system), the round robin approach gives a satisfactory spread of data in the input memories of the processing elements. This does not involve complex hardware as the only information that has to be stored by the controller is the last bus to be used. However the allocation of processes to processing elements is not so simple: it has been shown that a scheduling algorithm which takes into account the "busy-ness" factor of individual processing elements gives a more efficient system than one in which processes are randomly allocated. If this is to be incorporated into the hardware design it involves the passage of data from processing elements to the controller at regular intervals, this data being used to update a central store of information. In addition the controller has to consult this store of information each time a process is allocated to a processing element. Possible hardware mechanisms for realising this function are discussed in [Brown 89], the important consideration here is that the whole question of efficient process scheduling is a difficult one and the use of sophisticated hardware to implement it may not prove cost effective. The test results discussed in Chapter 9.6.3 show that even where an efficient method of implementing a "busy-ness" factor for each processing element can be devised, this may not act as a reliable prediction as to the amount of actual processing involved in the execution of waiting processes. Because of wide discrepancies in processing times for individual processes it is difficult to devise a meaningful measure to be used for load balancing. This would appear to be a fundamental weakness in the process based approach to parallel logic language execution; other projects have attempted to minimise it by not allowing parallel execution to be initiated until a certain number of processes have been queued up for execution in one processing element (see Chapter 3.1.3.3). In the PLL system it is not clear whether this approach would offer any advantages. It may be that this form of unpredictability has to be accepted and other forms of load balancing attempted: the scheduling of processes on a round robin approach should be considered at some future stage as this would involve simpler hardware and obviate the need for information flow regarding the "busy-ness" from processing element to the controller. In general these design considerations should be delayed until a more efficient form of parallel PLL rewrite interpreter can be produced.

The question of memory utilisation in the processing elements is of importance. The architecture appears to provide an efficient system for parallel execution of independent OR processes and data transmission times are kept low by the definition of a data packet which can be received by many processing elements simultaneously. However it is an essential feature of this system that a considerable amount of static data, namely the inbuilt and user defined rewrite rules, is duplicated in each processing element. Other proposed parallel logic language systems have also incorporated this into the computational model and as the figures for rule storage requirements have indicated it may be realistic to implement this directly in respect of high level rules (see Chapter 6.4.4). However any realistic knowledge based system based on the parallel PLL will need to store base predicates on disk.

Thus the final aspect of the design of the hardware that has to be raised is the secondary storage of data and its transfer into the processing elements. It has been recognised that for a realistic system the efficiency of this is crucial to the overall performance but no detailed proposals have been put forward. The access to data on disk involves two considerations: how to link the disk units with the parallel machine, ie what data paths are to be provided, and how the data should be organised on disk, ie what form of indexing schemes should be applied. This is an important research area for many database projects: in situations where the processing of data is to be performed on a parallel machine different considerations may apply than those involved where multiple disk units are linked to a single processor machine [Gray 90b].

10.3. Research Methods and Project Organisation

10.3.1. Introduction

This section looks at the manner in which the project developed, the organisation of the work and the tools used to implement the parallel PLL system. The organisation of the tasks in the project followed the traditional development cycle, ie familiarisation with the area of concern, analysis and specification of the requirements for the system, followed by implementation and testing.

The original design and development of the sequential version of the Pure Logic Language was done by research staff at ICL, but the decision to use it as a basis for a parallel logic system was taken by the author of this thesis. There were two distinct aspects to the work on the parallel system: the work on the computational model for the PLL and the design of the architecture on which to run the system. The work on the design and realisation of the architecture was done by John Brown and is separately documented in [Brown 89]. He was responsible for the decision to implement the message passing mechanism defined for the parallel language system by the introduction of multiple broadcast busses into a specialised custom built multiprocessor machine. The work on the computational model for the parallel logic language, including the decision to implement an OR parallel process model, was undertaken by the author of the thesis as was the design and implementation of the simulation software.

Before looking at the progress of the project it is worth considering the problem of abstraction that has arisen throughout this project. The different levels of abstraction involved in the overall system design and implementation have at times been confusing and do not make the task of defining and describing the system easy. Three components of the system can be identified: the abstractions involved in the definition and implementation of the process concept in the language system (Fig.6.7), and in the mapping of the parallel interpreter onto a single processor system (Fig.10.1) and finally the the modelling and simulation of the multiprocessor machine design (Fig.10.2.) It is hoped that the interface between the different levels in each component has been clearly identified and described, and that the links between them are unambiguous.

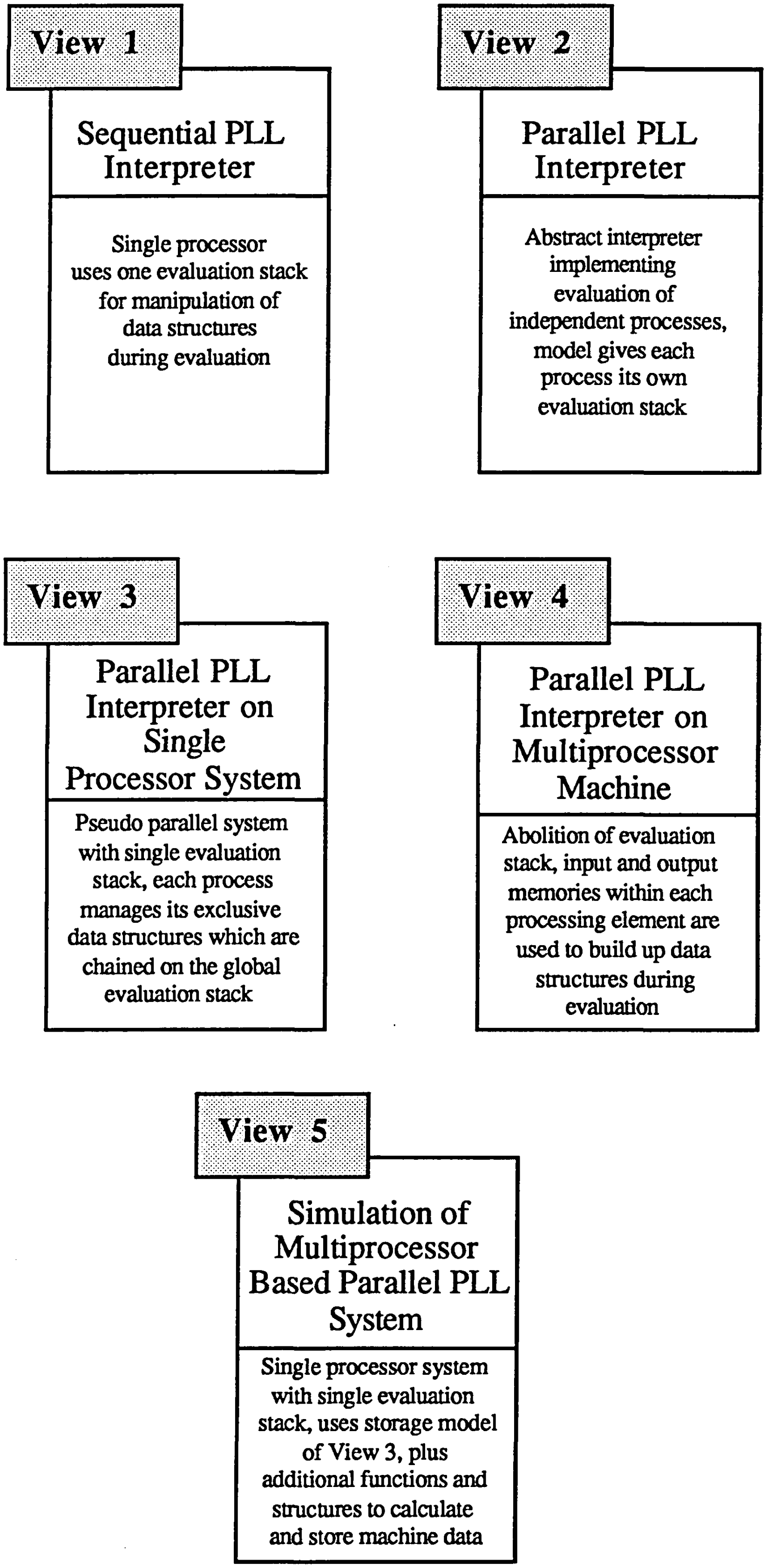


Fig. 10.1 - Representation of Different PLL Memory Mapping Views

10.3.2. Background Work

The impetus to the work on a parallel version of the PLL came from previous work carried out at Sheffield City Polytechnic into the design of multiprocessor machines to implement data flow programs. This was extended to look at the suitability of parallel architectures for applications in the field of artificial intelligence, in particular semantic networks of the type proposed by Fahlman [Fahlman 79]. Thus the basic expertise in this area prior to this project lay in the field of knowledge representation and multiprocessor architectures rather than parallel logic languages. Because of this the process of familiarisation with the area of interest and the formulation of realistic goals for the project occupied a considerable period of time during the project's life cycle.

The background work carried out in the first stage of this project involved familiarisation with three broad areas: parallel architectures with particular emphasis on those designed for symbolic processing applications, knowledge representation and knowledge based systems including deductive databases, and finally logic languages including parallel logic language systems. This background work is documented in two reports: the first on parallel architectures and knowledge representation, and the second on automated theorem proving and parallel logic languages [Jelly 87], [Jelly 88]. This phase of the work which included the preparation of these reports occupied more than the first year of the project.

At the same time as the general literature review on parallel logic languages was being carried out the study of ICL's Pure Logic Language was started. This involved an evaluation of the first version of the interpreter which was written in LISP. Various test programs were devised to enable the potential of the language to be evaluated. It was clear that it showed a number of interesting features that separated it from Prolog and the decision was taken that it should form the basis for the exploration of parallelism in logic languages. This was encouraged by a co-operative relationship with ICL who were willing to supply the sequential interpreter and internal research papers.

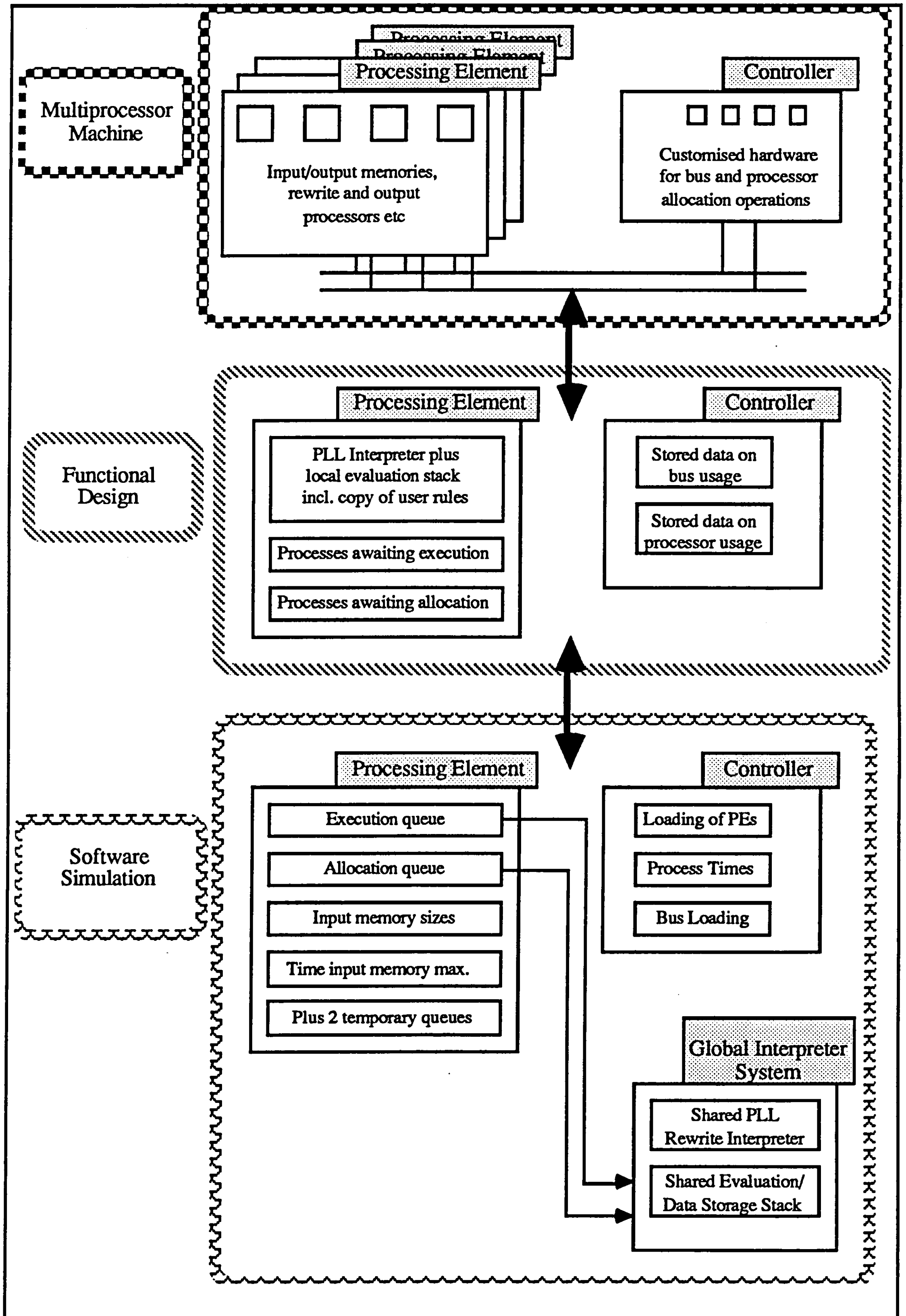


Fig. 10.2 - Simulating the Parallel Machine Design

10.3.3. Analysis and Specification of the System Requirements.

Before a specification of the parallel PLL system and architecture could be proposed a detailed study of the PLL was required. A new version of the system was obtained: this was now written in C and ran on the Archimedes microcomputer and the Sun workstation. Both machines have been used in the course of the project, Chapter 10.4 considers their use and the subsequent move to the Transputer based system. At this stage a detailed understanding of the new interpreter was required and this involved the design of further tests and a lengthy period of code inspection and documentation. This was prolonged by the absence of program documentation for the ICL interpreter.

The decision to implement OR parallelism and omit any form of AND parallelism was taken after a considerable amount of work had been done on the possibility of including both forms of parallelism. It would appear from the results given in Chapter 8 that this decision was justified and once it had been taken the requirements for the computational model of the PLL could be formulated relatively easily.

The functional requirements for the multiprocessor architecture evolved from the model for the parallel logic language system. The decision to encapsulate the one to many process spawning operation in a broadcast mechanism looked attractive and a machine design which could realise this was developed. The technical feasibility of the design is considered in [Brown 89].

The specification of the software system which was needed to implement the parallel language and the machine architecture reflected its dual nature. The requirements of the simulation model were specified by providing a high level description of the machine structures and operations that had to be included, and listing the information that the simulation was required to produce. The program design was constructed to "match" the machine structures as has been discussed in Chapter 7.4, and operations specified that would emulate the "real" machine's functioning in order to ensure as close a fit between the simulation and design as possible. No attempt was made to provide a formal specification of the simulation.

The identification and specification of the requirements for the new interpreter involved the introduction of the concept of process based execution. Because the basic system, ie the sequential interpreter, was already in existence, the definition of a process involved identifying the appropriate sequence of processing and linking it in an unambiguous manner with the abstract notion of "process".

The processing tasks performed by the sequential system that were redefined as a "process" effectively involved all the rule rewriting operations that took place from the start of rewriting until either no further rewrites were possible or an OR node was encountered. In the latter case the creation of new processes was implemented before the parental process terminated. Having specified the requirements for the definition of a process the question of implementation was considered.

10.3.4. Implementation

As has been discussed previously the implementations of the two distinct parts of the system, ie the parallel version of the interpreter and the machine simulation, were carried out concurrently. The approach taken was to define an interface between the two components: this interface would reflect the functional requirements of the system as well as making total separation of the components easy should this be required. This interface was chosen to be the process structure (see Chapter 5.4.3). The interpreter system "executes" a process structure using the information contained in the process description part of the structure, the machine simulation used the control information contained in the structure to manipulate the storage of processes.

Having designed this basic linking structure the simulation software was developed separately: the structures and functions which had been defined at the specification stage for the parallel machine simulation were realised in code. These machine components handled the manipulation of process structures, where appropriate passing them to the parallel interpreter for execution. Initially this was a "dummy" interpreter which "executed" a process by destroying it and spawning a random number of offspring processes, this number being in the range zero to ten. These dummy processes contained no process description but held the appropriate control information (including arbitrary times for the start and finish of each process) that allowed the machine simulation to manipulate them. It was found that this was a very effective and safe method of producing the software to model the machine operation: it allowed the main design features of the simulation to be installed without the complexities of the parallel interpreter becoming involved.

The next step after the coding of a simulated machine which worked for dummy processes was the implementation of the parallel version of the PLL interpreter. This involved a respecification of the inbuilt system rewrite rules. Most of these rules required some form of alterations and others needed completely new versions. It was decided to incorporate the parallel version in a general program which also held the sequential version. This was done as an aid to program development and testing: it allowed the parallel and sequential modes of operation to be directly compared with each other.

10.3.5. Testing

The primary purpose for developing a simulation of the parallel PLL running on the proposed multiprocessor architecture was to obtain data on the predicted performance of the system. This involved the timing of many of the aspects of the system's behaviour during the testing phase. These timings were of two types: timings for the simulated behaviour of the parallel hardware (data transmission times, delays obtaining busses etc) and those for the execution of processes in the parallel interpreter. The former type of timings had of necessity to be estimated, there being no "real" parallel machine. The interpreter however did exist and the intention was to use the inbuilt system clock of the computer running the system in order to obtain absolute times. This proved more difficult than was anticipated and necessitated a move to the Transputer based system, the full implications of this are considered in Chapter 10.4.

In order to obtain the required data on system performance two initial tasks were necessary: the writing of suitable test programs and the development of a software package to interpret and display the results. The implementation of the results evaluation program was a straightforward task but production of suitable PLL programs to act as benchmarks was hampered by space restrictions in the Transputer system. The resulting test programs have been described in Chapter 8.3 and it is recognised that although they allow a reasonable amount of OR parallel activity to be defined and measured, the system would have to be expanded for any future testing.

10.4. Assessment of Programming Environments

The ICL Pure Logic Language interpreter used in this project was written in C and therefore it made sense to develop the parallel version and the machine simulation in the same language. The C language is frequently used to implement interpreters and compilers for high level languages: the features it offers are very suitable for this type of task and most implementations of C give good performance figures [Kernighan 78]. The conversion of the specified data structures defined for the machine simulation into C data structures was straightforward and C functions were produced to match the operations required in the machine.

The use of C was beneficial in that it proved highly portable. The parallel PLL system has been developed on three separate computers with different compilers, libraries and other system tools. However no problems of code compatibility have arisen during these transfers except in a small number of clearly recognised places where library functions specific to the target machine have been used, eg the Transputer "timer_now()" C function [3L Parallel C 88].

Three different computers and programming environments were used during the project's life time. The initial stage of the work on the simulation and the parallel interpreter was performed using an Archimedes M310 microcomputer. This machine was also used at ICL and allowed easy exchange of software. However there were two drawbacks to the Archimedes system. The first problem related to the fact that it was a new system and the first version of the operating system was not working correctly. This was eventually replaced by a better version which although more secure was not "bug-free"; the problem was compounded by the lack of good documentation. However the C libraries and compiler for the Archimedes appeared to be much more secure and code was compiled and linked very quickly. (The operating system problems with the Archimedes series of computers have hopefully been solved by the introduction of the RISCOS multitasking system which is now in general use). The second drawback to the Archimedes system was that the memory (1 Mbyte) proved insufficient to hold the whole of the parallel PLL system including the machine simulation and leave enough space to allow test programs to be executed. The manner in which the interpreter worked was to allocate any free space remaining once code had been installed to implement the execution stack for the PLL. Under the Archimedes configuration the PLL execution stack was too small to allow any major rewriting to take place in it.

The partially developed simulation/parallel PLL interpreter code was transferred and recompiled on the Sun 3/60 workstation which had 4 Mbytes of memory. This Unix based system proved ideal for program development: the SunView windowing facility adds to the ease and speed of programming as it allows the programmer to view the source code and execution paths of a program simultaneously.

Unfortunately problems arose with the Sun system when the installation of code to implement the timings of process execution were introduced. Initially the system had operated with arbitrary values for process timings in anticipation of using the Sun system clock at a later stage. However when this was attempted it became clear that the clock granularity of 16 ms would not provide the information required. Most processes had total execution times of less than this and it was also desirable to be able to time subparts of each process's execution, eg the "set up", "rewrite" and "spawn" times as defined in Chapter 8.3.2.

The availability of a C compiler for a Transputer based system meant that the decision was taken to port the software onto this system which consisted of a host computer, the Tandon PCA-20, with a T414 Transputer board with 2 Mbytes of external memory. The use of a PC clone provided a much less attractive environment for program development and the total amount of available memory was reduced in comparison with the Sun. However there was sufficient space to run the test programs as given in Appendix C and this was acceptable for the present series of tests. The benefit was that timings in units of 1 microsec were now available from the Transputer clock. The obtaining of valid timing results was complicated by the architecture of the system, ie the division into on-chip and external RAM. Familiarity with the method of configuring the code and workspace with respect to the two types of memory and the implications for the timing of programs took time to develop. This is documented in Appendix J.

The move to the Transputer allowed real times to be obtained for the execution of processes which gives an authority to the overall test results. The unusual Transputer architecture was a drawback to obtaining good comparative data: the feature which gives the Transputer much of its performance advantage as a processor, ie the on-chip RAM, was a disadvantage for these tests. A full working version of the interpreter could utilise the internal RAM to improve performance and this is an area for further experimentation to decide on optimum code and workspace placement (see Appendix J).

10.5. Future Work

At various stages throughout the thesis suggestions have been made for further work in the area of the parallel execution of the PLL and the design of a multiprocessor system for it. This section identifies the important areas for future research and summarises the tasks. Broadly this work can be considered as relating to the PLL rewrite interpreter or to the architectural proposals.

The present parallel PLL interpreter has developed out of work by ICL on the sequential version but both systems can be regarded as experimental prototypes. In moving towards more realistic implementations consideration must be given to improving the efficiency of both versions.

The method used in implementing conjunction rewriting is fundamental to the operation of the Pure Logic Language; the elimination of order sensitivity as seen in Prolog is a major strength of the system but its implementation involves an unpredictable amount of repeated computation as described in Chapter 4.5.2 and Chapter 4.7. Repeated computation takes place because of the need to return to "earlier" subexpressions when bindings are made during the evaluation of "later" subexpressions.

Essentially rewriting of conjoined expressions can be seen as an ordering problem: if the subexpressions can be evaluated in a manner which binds variables in a logical order there will be no need to return to "earlier" subexpressions to repeat their evaluation. This situation is directly analogous to that existing with the ordering of subexpressions for parallel execution in an AND parallel system. As described in Chapter 2.3.4.4 and Chapter 3.1.2.2 this can be achieved by the inclusion of data dependency analysis techniques in order to allow "safe" parallel execution to take place. This "safe" execution is guaranteed when the pattern of shared variable instantiation follows an ordered approach with the first subexpression to execute acting as "producer" and subsequent ones designated "consumers". With conjunction rewriting in the PLL it would appear valuable to order subexpressions in such a manner as to allow "safe" sequential evaluation to take place, but the criteria for the determination of "safety" are identical in both the parallel and sequential cases. Thus it is hoped that the introduction of techniques evolved for AND parallel data analysis into the PLL rewrite interpreter could prove of considerable value in optimising the sequential process of conjunction rewriting. The details of this would almost certainly involve some rule compile time analysis with run time checking of the pattern of variable instantiation.

The introduction of some form of variable analysis would be appropriate for both the sequential and the parallel systems. It is not clear how the existence of OR parallelism within the system would affect the run time marking of variable binding patterns. It is likely that some additional information would need to be conveyed with the broadcast data packet but this area has to be subject to further investigation.

The question of the design of the coding for the rewrite interpreter has been raised in Chapter 8.5.4 and Chapter 9.4.4 where a detailed evaluation of the function calling overheads in the system was made. The conclusion from those series of tests was that there was room for considerable improvement in performance by a reorganisation of the code design. This issue is indirectly linked to the suggestion for the introduction of a variable dependency scheme for conjunction rewriting. If the handling of conjunction rewriting can be altered to give a more deterministic pattern to the evaluation of a conjoined expression, the abandoning of the recursive basis of the rewriting algorithm is easier and the move to a fully compiled system becomes a more realistic possibility.

At present the state of the evolving query is represented by an expression tree held on the evaluation stack. Future work should address the question of whether the organisation of the data structures used to control query evaluation is the most effective. A move towards a compiled system would also involve low level optimisations of the type seen in the Warren Abstract Machine including the use of registers to hold frequently used pointers and values.

The possibilities for future work so far considered are equally applicable to the sequential and parallel PLL systems. Future research into the parallel proposals put forward in this thesis can be divided into two types: work directly related to the multiprocessor architecture, and that concerned with the parallel computation model. The crucial feature which differentiates the parallel PLL system from other parallel logic language proposals is the incorporation of the broadcasting approach within the interpreter and the proposals for implementing it. However the present system exists only in the form of a simulation: not only has no "real" parallel implementation of the hardware been attempted but the "parallel" interpreter as now constructed has been designed and written to run on a single processor. Thus before any attempt to port the system onto parallel hardware further work on the parallel interpreter is necessary.

The discussion on Chapter 8.5.4 has shown that there is room for considerable improvement in the efficiency of the software responsible for process spawning. This needs to be recoded to implement correct data packet creation and to improve the performance of the spawning operation.

The other obvious area to investigate before moving towards a full scale parallel implementation is the use of Prolog in the system. This would involve the adaptation of Prolog to fit the OR parallel process computational model and the installation of code to implement the type of process spawning used with the PLL. From the theoretical viewpoint this raises issues of handling Prolog's extra logical features, eg the cut, assert and retract operations. There has been a considerable amount of work on the inclusion of side effects in parallel systems and this would have to be looked at in detail in relation to a Prolog system based on broadcasting of spawned processes [Kale 88b].

Future work on the architectural side should consider the feasibility of prototype building. The design for the broadcast bus based multiprocessor machine as proposed in Chapter 6.5 involves a considerable amount of customised hardware [Brown 89]. However the first step is to present a hardware model of the system implemented in currently available technology. The crucial aspect of the design of an appropriate prototype is the communication systems which involve links between the processing elements and the controller, and between the processing elements themselves. These systems in the multiprocessor design can be considered as implementing three different tasks:

a) the setting up of data packet transfer; this is conceptually a point to point communication between processing elements and the controller,

b) data packet transfer; the broadcast operation between one processing element and many others,

c) return of results; this is a one to one communication as defined in a).

In order to model the architecture using standard technology these three communication systems would have to be implemented in such a manner as to allow their individual performances to be separately monitored. This approach would not necessarily produce a high performance prototype but it would ensure that each aspect of the communication network design was tested and would thus contribute useful information to the detailed design of the final machine.

Finally on the architectural side the storage and manipulation of base predicates on disk should be addressed. Multiple paths from the processing elements to disk units can be provided and this raises queries about the optimum organisation of the data.

10.6. Summary

This chapter has attempted to evaluate the project in two ways: first by presenting a critical review of the work on the parallel PLL interpreter and its proposed multiprocessor architecture, and secondly by looking at the development and organisation of the work throughout the project's duration. Finally suggestions for future research in this area have been made.

Chapter Eleven

Conclusion

The aim of this project was to investigate the parallel execution of logic programming languages and to relate this to architectural considerations for the design of multiprocessor machines. The thesis has presented the details of the resulting parallel logic language system and associated machine design.

The basis of the exploration of parallelism within logic languages has been the Pure Logic Language. This language system is of importance because it represents a practical approach to the execution of "pure" logic based on an interpreter which can be viewed as a set of rewrite rules. In order to maintain the semantics of the language it was recognised at an early stage that any move towards a parallel execution model for the PLL had to incorporate the notion of "automatic" parallelism, ie programmer control of execution was unacceptable.

Focusing on the style of logic program used in knowledge based systems allowed decisions to be made on the form of parallelism to be incorporated in the computational model. This type of non deterministic Datalog program shows good potential for performance improvements within an OR parallel scheme, whereas it is doubtful that there would be substantial benefit from the introduction of AND parallelism.

These two aspects, the automatic control of parallel execution and the use of OR parallelism, led to the proposals for an abstract computational model for the PLL. The model incorporated a third concept: it had to allow for the implementation of the system on a non shared memory machine. Work on the design of parallel machines had indicated that, because of scalablity problems, the project should not concern itself with the design of a shared memory machine. Thus the computational model for the parallel PLL was based on the evaluation of independent OR processes which communicated by message passing. These messages represented the information required for a process to initiate execution and to run to completion without further communication. The computational model was realised in the form of a new parallel rewrite interpreter.

The aspect which separates the approach taken in this project from other work in the field is the recognition of the spawning activity of processes as displaying a one to many pattern and the implementation of this in a generalised broadcast operation. Instead of a parent process sending individual messages to offspring processes, the information required by the children is conveyed in one broadcast package. This means that conceptually communication overheads are not dependent on the number of offspring processes but on parental processes. In a program showing a large degree of non determinism this is likely to result in considerable savings.

On the architectural front work on parallel architectures has resulted in proposals for the design of a novel multiprocessor machine which incorporates a mechanism to allow the broadcasting of messages to take place in an efficient and flexible manner. In order to obtain predictive performance indicators, a software simulation of the architecture has been written and the new parallel rewrite interpreter mapped onto it. The resulting software system has produced a large amount of information on the performance of the interpreter and aspects of machine functioning. These have enabled a detailed evaluation of the proposals to be made and led to suggestions for future work.

The research presented in the thesis has made a useful contribution towards the implementation of parallel logic languages by the investigation of the use of broadcasting. If broadcasting can be efficiently implemented it has been shown that a system can be defined in which there is no major overhead for process creation or spawning. This theoretically allows the full amount of available parallelism to be exploited, the only constraints being limitations on the numbers of available processing elements and thus load balancing considerations.

Finally the importance of parallel execution in the field of artificial intelligence and deductive databases is increasingly recognised and is the focus of a considerable volume of research. Although this project has concerned itself exclusively with the use of logic languages as a means of implementing certain types of programs, the concept of process based execution and message passing by broadcasting is not specific to this programming paradigm. It is hoped that the work contained in this thesis may prove of value in the wider context of knowledge based systems.

Bibliography

[3L Parallel C 88] **Parallel C Users Guide,** 3L Ltd, 1988.

[Addis 90] **CARDS: A Large Knowledge-Base Engine,** T.Addis, M.Nowell, Proc. Colloquium on Very Large Knowledge-Based Systems, IEE, London, June 1990.

[Ali 88a] **OR-Parallel Execution of Prolog on BC-Machine,** K.Ali, Int. Report, Swedish Institute of Computer Science, Kista, Sweden, May 1988.

[Ali 88b] **OR-Parallel Execution of Prolog on BC-Machine,** K.Ali, Proc. Fifth Int. Conf. on Logic Programming, Seattle, Washington, Aug. 1988.

[Alshawi 88] **The Delphi Model and Some Preliminary Experiments,** H.Alshawi, D.Moran, Proc. Fifth Int. Conf. on Logic Programming, Seattle, 1988.

[Anderson 87] **The Architecture of FAIM-1,** J.Anderson, W.Coates, A.Davis, R.Hon, I.Robinson, S.Robison, K.Stevens, IEEE Computer, Jan. 1987.

[Avramov 90] **Evaluation of Two Systems for Distributed Message Passing in Transputer Networks,** N.Avramov, A.Knowles, Proc. 13th Occam User Group Meeting, York, Sept. 1990.

[Babb 83] **Finite Computation Principle: An Alternative Method of Adapting Resolution for Logic Programming,** E..Babb, Proc. Int. Conf. on Logic Programming, Algarve, 1983.

[Babb 86a] **Towards the Exact Execution of Logic,** E.Babb, Int. Report, SSC, ICL, Oct. 1986.

[Babb 86b] **Mathematical Logic in the Large Practical World,** Int. Report, SSC, ICL, Oct. 1986.

[Babb 87] **Execution of Pure Logic,** E.Babb, I.Nairn, D.Cooper, P.Slessinger, Report for Alvey Exhibition, Manchester, July, 1987.

[Babb 89a] **Note on Extending PLL,** E.Babb, Int. Report, SSC, ICL, Feb. 1989

[Babb 89b] **Pure Logic Language,** E.Babb, ICL Technical Journal, May, 1989.

[Bancilhon 86] **An Amateur's Introduction to Recursive Query Processing Strategies,** F.Bancilhon, R.Ramakrishnan, ACM SIGMOD '86, Washington, DC, May 1986.

[Beer 86] **The German Parallel Prolog Machine Development Project,** J.Beer, in **Fifth Generation Computer Architectures,** Ed. Woods, Elsevier Science Publishers B.V.(North-Holland), 1986.

[Bhuyan 89] **Performance of Multiprocessor Interconnection Networks,** L.Bhuyan, Q.Yang, D.Agrawal, IEEE Computer, Feb. 1989.

[Biswas 88] **A Scalable Abstract Machine Model to Support Limited-OR (LOR)/Restricted-AND Parallelism (RAP),** P.Biswas, S-C Su, D.Yun, Proc. Fifth Int. Conf. on Logic Programming, Seattle, Washington, Aug. 1988.

[Bosco 89] **IDEAL & K-LEAF Implementation: a progress report,** P.Bosco, C.Cecchi, C.Moiso, Proc. PARLE 89 (Parallel Architectures and Languages Europe), Eindhoven, June 1989.

[Brachman 85] **On the Epistemological Status of Semantic Networks,** R.Brachman, in **Readings in Knowledge Representation,** Ed. Brachman, Morgan Kaufmann, 1985.

[Bratko 86] **Prolog Programming for Artificial Intelligence,** I.Bratko, Addison Wesley, 1986.

[Brown 87] **Taxonomic Hierarchies with Attribute Inheritance: Potential and Realisable Parallelism,** J.Brown, Proc. First Colloquium of Mathematics and Computing, Univ. of Bordeaux/Weizmann Institute, Bordeaux, Dec. 1987.

[Brown 88] **Taxonomic Hierarchies with Attribute Inheritance: Potential and Realisable Parallelism [Extended Version],** Proc. Conf. on Parallel Architectures for AI, BCS/Occam Users Group, London, Feb. 1988.

[Brown 89] **Parallel Architectures for IKBS,** J.Brown, Int. Report, Dept of Computing, Bradford Univ, 1989.

[Bundy 83] **The Computer Modelling of Mathematical Reasoning,** A.Bundy, Academic Press, 1983.

[Butler 88] **Scheduling OR Parallelism: An Argonne Perspective,** R.Butler, T.Disz, E.Lusk, R.Olson, R.Overbeek, R.Stevens, Proc. Fifth Int. Conf. on Logic Programming, Seattle, Washington, Aug. 1988.

[Calderwood 88] **Scheduling OR-Parallelism in Aurora - the Manchester Scheduler,** A.Calderwood, P.Szeredi, Int. Report, Dept. of Computer Science, Manchester University, 1988.

[Campbell 84] **Implementations of Prolog,** J.Campbell, Ellis Horwood, 1984.

[Capon 86] **PARSIFAL - A Parallel Simulation Facility,** P.Capon, J.Gurd, A.Knowles, Occam Users' Newsletter, No.5, July 1986.

[Carlsson 88] **A Simplified Approach to the Implementation of AND Parallelism in an OR Parallel Environment,** M.Carlsson, K.Danhof, R.Overbeek, Proc. Fifth Int. Conf. on Logic Programming, Seattle, Washington, Aug. 1988.

[Chang 78] **DEDUCE 2: Further Investigations of Deduction in Relational Databases,** C.Chang, in **Logic and Databases,** Ed. Gallaire, Plenum Press, 1978.

[Chang 85] **AND Parallelism of Logic Programs Based on a Static Data Dependency Analysis,** J.Chang, A.Despain, D.DeGroot, Proc.IEEE Compcon, Feb.1985.

[Charniak 85] **Introduction to Artificial Intelligence,** E Charniak, D McDermott, Addison Wesley, 1985.

[Chassin de Kergommeaux 88]
An Abstract Machine to Implement Efficient OR/AND Parallel Prolog, J.Chassin de Kergommeaux, P.Robert, Addendum to Proc. Int. Conf. Logic Programming, Seattle, Washington, 1988.

[Chassin de Kergommeaux 89]
Performance Analysis of a Parallel Prolog: A Correlated Approach, J.Chassin de Kergommeaux, U.Baron, W.Rapp, M.Ratcliffe, Proc. PARLE '89, Eindhoven, 1989.

[Cheese 87a] **A Parallel Model of Inference for Knowledge Bases,** A.Cheese, Proc. Workshop of Special Interest Group in Knowledge Manipulation Engines, Brunel Univ, 1987.

[Cheese 87b] **A Concurrent Architecture for Committed-Choice Non-Deterministic Logic Languages,** A.Cheese, Research Report, Dept. Computer Science, Univ. of Nottingham, 1987.

[Ciepielewski 86] **Initial Evaluation of a Virtual Machine for OR-Parallel Execution of Logic Programs,** A.Ciepielewski, B.Hausman, S.Haridi, in **Fifth Generation Computer Architectures,** Ed. Woods, Elsevier Science Publishers B.V.(North-Holland), 1986.

[Clark 82] **Logic Programming,** K.Clark, S-A. Tarnlund, Academic Press, 1982.

[Clark 83] **PARLOG: A Parallel Logic Programming Language, K.Clark, S.Gregory,** Research Report DOC 83/5, Imperial College, Univ. of London, 1983.

[Clark 86] **PARLOG: Parallel Programming in Logic,** K.Clark, S.Gregory, Trans.ACM (Programming Languages and Systems), Vol.8, No.1, Jan.1986.

[Clocksin 81] **Programming in Prolog,** W.Clocksin, C.Mellish, Springer-Verlag, 1981.

[Codd 71] **A Relational Model of Data for Large Shared Data Banks,** E.Codd, Comm.ACM, 13, 1971.

[Colmerauer 73] **Une System de Communication Homme-Machine en Francais,** A.Colmerauer, H.Kanoui, P.Roussel, R.Pasero, Groupe de Recherche en Intelligence Artificielle, Universite d'Aix-Marseille, 1973.

[Conery 83] **The AND/OR Process Model for Parallel Interpretation of Logic Programs,** Tech Report 204 (PhD Thesis), Univ California, Irvine, 1983.

[Conery 85] **AND Parallelism and Nondeterminism in Logic Programs,** J.Conery, D.Kibler, New Generation Computing, Vol.3, 1985.

[Conery 87] **Binding Environment for Parallel Logic Programming in Non-Shared Memory Multiprocessors,** J.Conery, Proc. Symposium on Logic Programming, San Francisco, Sept. 1987.

[Cooper 87a] **Incorporation of More Complex "AND" Theorems into the Pure Logic Language,** D.Cooper, Int.Report, SSC, ICL, April 1987.

[Cooper 87b] **A Computational Model for the Pure Logic Language,** D.Cooper, Int.Report, SSC, ICL, Oct.1987.

[Cooper 87c] **Translation of Pure Prolog Programs into the Pure Logic Language,** D.Cooper, Int.Report, SSC, ICL, Oct.1987.

[Dahl 83] **Logic Programming as a Representation of Knowledge,** V.Dahl, IEEE Computer, Oct.1983.

[Darlington 87] **Notes from Research Seminar,** J.Darlington, Univ. of Sheffield, 1987.

[Date 85] **An Introduction to Database Systems,** Fourth Edition, C.Date, Addison-Wesley, 1985.

[DeGroot 84] **Restricted AND Parallelism,** D.DeGroot, Proc. Int. Conf. on Fifth Generation Computing Systems, Nov.1984.

[DeGroot 87] **Restricted AND Parallelism and Side Effects,** D.DeGroot, Int. Symposium on Logic Programming, San Francisco, 1987.

[Diel 86] **Parallel Logic Programming Based on an Extended machine Architecture,** H.Diel, in **Fifth Generation Computer Architectures,** Ed. Wood, Elsevier Science Publishers B.V. (North Holland), 1986.

[Dijkstra 68] **Cooperating Sequential Processes,** E.Dijkstra, in Programming Languages, Ed Genuys, Academic Press, 1968.

[Fahlman 79] **NETL: A System for Representing and Using Real-World Knowledge,** S.Fahlman, MIT Press, 1979.

[Frost 86] **Introduction to Knowledge Based Systems,** R.Frost, Collins, 1986.

[Gallaire 78] **Logic and Databases,** H.Gallaire, J.Minker, Plenum Press, 1978.

[Gallaire 84] **Logic and Databases: A Deductive Approach,** H.Gallaire, J.Minker, J-M.Nicolas, ACM Computing Surveys, Vol.16, No.2, June 1984.

[Gardarin 89] **Relational Databases and Knowledge Bases,** G.Gardarin, P.Valduriez, Addison Wesley, 1989.

[Gray 84] **Logic, Algebra and Databases,** P.Gray, Ellis Horwood, 1984.

[Gray 85] **Efficient Prolog Access to Codasyl and FDM Databases,** P.Gray, Proc. ACM SIGMOD, 1985.

[Gray 87a] **A Prolog Extension to the Functional Data Model with Modular Commitment,** P.Gray, Proc. Workshop of Special Interest Group in Knowledge Manipulation Engines, Brunel Univ, May 1987.

[Gray 87b] **Comparison of Hardware Assistance to Database and IKBS,** P.Gray, Proc. Workshop of Special Interest Group in Knowledge Manipulation Engines, Brunel University, May 1987.

[Gray 90a] **An Object Orientated Approach to a Parallel Database System,** J.Gray, Proc. Seminar Series on New Directions in Software Development, Ed. Robinson, Wolverhampton Polytechnic, 1990, (also to appear in revised form in the journal Information and Software Technology).

[Gray 90b] **Parallel-DB4GL, A Transputer Based Implementation of a Parallel Database System,** J.Gray, Proc. Brit. Nat. Conf. on Databases, York, July 90.

[Haridi 89] **Cache Coherence Protocol of the Data Diffusion Machine,** S.Haridi, E.Hagerston, Proc. PARLE '89, Eindhoven, 1989.

[Harrison 86] **Parallel Architectures for Non-Sequential Programming Systems,** P.Harrison, M.Reeve, Int. Report, Dept. of Computing, Imperial College, Univ. of London, 1986.

[Hausman 89] **Pruning and Scheduling Speculative Work in OR Parallel Prolog,** B.Hausman, Proc. PARLE '89, Eindhoven, 1989.

[Hillis 85] **The Connection Machine,** D.Hillis, MIT Press, 1985.

[Hillis 86] **Data Parallel Algorithms,** D.Hillis, G.Steele, Comm. ACM, Dec. 1986.

[Hird 85] **Modifications to a Data Flow Multiprocessor for Artificial Intelligence Applications,** B.Hird, B.Sc.Project Report, Sheffield City Polytechnic, 1985.

[Hogger 84] **Introduction to Logic Programming,** C.Hogger, Academic Press, 1984.

[Hughes 86] **PARSIFAL Project Summary for the Alvey Architecture Club,** D.Hughes, April, 1986.

[Howarth 85] **The CAFS System Today and Tomorrow,** G.Howarth, ICL Technical Journal, Nov.1985.

[Hoare 78] **Communicating Sequential Processes,** A.Hoare, Comm.ACM, 21, 1978.

[Hwang 85] **Multiprocessor Supercomputers for Scientific/Engineering Applications,** K.Hwang, IEEE Computer, 1985

[Hwang 87] **Computer Architectures for Artificial Intelligence Processing,** K.Hwang, J.Ghosh, R.Chowkwanyun, IEEE Computer, Jan.1987.

[Hwang 89] **A Compiling Approach to Exploiting AND Parallelism in Parallel Logic Programming Systems,** Z.Hwang, S.Hu, Proc. PARLE '89, Eindhoven, 1989

[IF Prolog 88] **IF Prolog Compiler/Interpreter,** Intellicorp, in Catalyst, Sun Microsystems Inc, 1988.

[INMOS 88] **Occam 2 Reference Manual,** INMOS Ltd, Prentice Hall, 1988.

[INMOS 89] **The Transputer Databook,** Second Edition, INMOS Ltd, 1989.

[Intel 86] **Intel IPSC-VX Product Description,** Intel, 1986.

[Ito 83] **Parallel Prolog Machine Based on the Data Flow Model,** N.Ito, K.Masuda, H.Shimizu, TR-035, ICOT, 1983.

[Jelly 87] **First Report on Logic Languages and Knowledge Based Systems on Multiprocessor Architectures,** I.Jelly, Report to Alvey Directorate, School of Computing and Management Sciences, Sheffield City Polytechnic, May 1987.

[Jelly 88] **Second Report on Logic Languages and Knowledge Based Systems on Multiprocessor Architectures,** I.Jelly, Report to Alvey Directorate, School of Computing and Management Sciences, Sheffield City Polytechnic, Jan 1988.

[Kale 88a] **Parallel Execution Schemes,** L.Kale, Tutorial No.4, Fifth Int. Conf. on Logic Programming, Seattle, Washington, Aug.1988.

[Kale 88b] **A Memory Independent Binding Environment for AND and OR Parallel Execution of Logic Programs,** L.Kale, B.Ramkumar, W.Shu, Proc. Fifth Int. Conf. on Logic Programming, Seattle, Washington, Aug. 1988.

[KEE 86] **KEE Software Development System,** User's Manual, Intellicorp, 1986.

[Kernighan 78] **The C Programming Language,** B.Kernighan, D.Ritchie, Prentice-Hall, 1978.

[Kowalski 74] **Predicate Logic as a Programming Language,** R.Kowalski, Proc. IFIP, 1974.

[Lloyd 81] **Implementing Clause Indexing in Deductive Database Systems,** J.Lloyd, Research Report 81/4, Dept of Computer Science, Univ of Melbourne, Australia.

[Lloyd 84] **Foundations of Logic Programming,** J.Lloyd, Springer-Verlag, 1984.

[Loh 82] **A Data Flow Parallel Processor Consisting of a Rectangular Array of Processor Nodes,** M.Loh, J.Brown, Conf. on Twenty Five Years of Computing, Univ. of Leeds, 1982.

[Lusk 88] **The Aurora OR Parallel Prolog System,** E.Lusk, R.Butler, T.Disz, R.Olson, R.Overbeek, R.Stevens, D.H.D.Warren, A.Calderwood, P.Szeredi, S.Haridi, P.Brand, M.Carlsson, A.Ciepielewski, B.Hausman, Tech.Report, Argonne Nat. Labs/Manchester University/SICS,Stockholm, May 1988.

[Maher 88] **The Equivalence of Logic Programs,** M.Maher, in **Foundations of Deductive Databases and Logic Programming,** Ed Minker, Morgan Kaufmann, 1988.

[Maier 88] **Computing with Logic,** D.Maier, D.S.Warren, Benjamin/Cummings, 1988.

[May 88] **The Influence of VLSI Technology on Computer Architectures,** D.May, BCS/Occam Users Group Conf. on Parallel Architectures for A.I., London, Feb. 1988.

[McBrien 88a] **PLL User Guide,** Version 0.2, Issue A, P.McBrien, Int. Report, SSC, ICL, March 1988.

[McBrien 88b] **PLL User Guide,** Version 0.32, Issue A, P.McBrien, Int. Report, SSC, ICL, Oct. 1988.

[McBrien 88c] **Personal Communication,** P.McBrien, SSC, ICL, Oct.1988.

[MacDougall 87] **Simulating Computer Systems; Techniques and Tools,** M.MacDougall, MIT Press, 1987.

[McGregor 90] **Distributed Object Orientated Knowledge Bases,** D.McGregor, Proc. Colloquium on Very Large Knowledge Based Systems, IEE, London, June 1990.

[Meyer 88] **Object Oriented Software Construction,** B.Meyer, Prentice Hall, 1988.

[Minker 88] **The Foundations of Deductive Databases and Logic Programming,** J Minker, Morgan Kaufmann, 1988.

[Minsky 74] **A Framework for Representing Knowledge,** M.Minsky, Research Memo.No.306 in Artificial Intelligence, MIT, 1974.

[Minsky 85] **A Framework for Representing Knowledge,** M.Minsky, in **Readings in Knowledge Representation,** Morgan Kaufmann, 1985.

[Moffat 86] **Interfacing Prolog to a Persistent Data Store,** D.Moffat, P.Gray, Third Int. Conf. on Logic Programming, London, 1986.

[Mudge 87] **Multiple Bus Architectures,** T.Mudge, J.Hayes, D.Winsor, IEEE Computer, June 1987.

[Nairn 87] **PLL Based on a Rewrite Technique,** I.Nairn, Int. Report, SSC, ICL, March 1987.

[Naish 88] **Parallelising Nu-Prolog,** L.Naish, Proc. Fifth Int. Conf. on Logic Programming, Seattle, Washington, Aug. 1988.

[Neelamkavil 87] **Computer Simulation and Modelling,** F.Neelamkavil, John Wiley, 1987.

[Odijk 86] **The Philips Object-Orientated Parallel Computer,** E.Odijk, Fifth Generation Computer Architectures, Ed. Woods, Elsevier Science Publishers B.V.(North-Holland), 1986.

[PARLE 89] Proc. PARLE 89 (Parallel Architectures and Languages Europe), Eindhoven, June 1989.

[Ramakrishnan 88] **Magic Templates: A Spellbinding Approach to Logic Programs,** R.Ramakrishnan, Proc. Fifth Int. Conf. on Logic Programming, Seattle, Washington, Aug. 1988.

[Rhaman 88] **Fully Distributed AND/OR Parallel Execution of Logic Programs,** P.Rhaman, E.Stark, Proc. Fifth Int. Conf. on Logic Programming, Seattle, Washington, Aug. 1988.

[Ratcliffe 87] **The PEPSys Parallel Logic Programming Language,** M.Ratcliffe, J-C Syre, Proc. Int. Joint Conf on Artificial Intelligence, Milan, 1987.

[Rawlings 87] **A Large Knowledge Based System for Molecular Biology,** C.Rawlings, K.Seifert, J.Saldana, Proc. Workshop of Special Interest Group in Knowledge Manipulation Engines, Reading Univ, Jan. 1987

[Rawlings 90] **Large Knowledge Based Applications in Molecular Biology and Genetics,** C.Rawlings, D.Clark, I.Archer, G.Barton, J.Saldana, Proc. Colloquium on Very Large Knowledge-Based Systems, IEE, London, June 1990.

[Reiter 78a] **On Closed World Databases,** R.Reiter, in **Logic and Databases,** Ed Gallaire, Plenum Press, 1978.

[Reiter 78b] **Deductive Question-Answering on Relational Data Bases,** R.Reiter, in **Logic and Databases,** Ed. Gallaire, Plenum Press, 1978.

[Renterghem 89] **Transputers for Industrial Applications,** P.Van Renterghem, Concurrency: Practise and Experience, Vol.1(12), Dec.1989.

[Rettberg 86] **Contention is No Obstacle to Shared Memory Multiprocessing,** R.Rettberg, R.Thomas, Comm ACM, Dec 1986.

[Reynolds 87a] **The Design and Implementation of an AND/OR Parallel Logic Language,** J.Reynolds, A.Beaumont, L.Spacek, Proc. Workshop of Special Interest Group in Knowledge Manipulation Engines, Reading Univ, Jan. 1987.

[Reynolds 87b] **BRAVE - An Architecture for Parallel Inference,** J.Reynolds, T.Beaumont, A.Cheng, S.Dalgado-Rannauro, Proc. Workshop of Special Interest Group in Knowledge Manipulation Engines, Brunel Univ, May 1987.

[Robinson 65] **A Machine Orientated Logic Based on the Resolution Principle,** JACM, 12, Jan.1965.

[Saeedi 90] **A Parallel Frame Based System for Knowledge Representation,** M.Saeedi, PhD Thesis (in preparation), Sheffield City Polytechnic, 1990.

[Schank 75] **Conceptual Information Processing,** R.Schank, Elsevier North Holland, 1975.

[Schank 77] **Scripts, Plans, Goals and Understanding,** R.Schank, R.Abelson, Erlbaum, 1977.

[Schwin 89] **RAPiD, A Dataflow Model for the Implementation of Parallelism and Intelligent Backtracking in Logic Programs,** B.Schwin, G.Borth, C.Welsh, Proc. PARLE 89 (Parallel Architectures and Languages Europe), Eindhoven, June 1989.

[Shapiro 83] **A Subset of Concurrent Prolog and Its Interpreter,** E.Shapiro, Tech. Report, Weizmann Institute of Science, Rehovet, Israel, 1983.

[Shapiro 86] **Concurrent Prolog: A Progress Report,** E.Shapiro, IEEE, Computer, Vol.19, No.8, Aug.1986.

[Shaw 85] **NON-VON's Applicabilty to Three AI Task Areas,** D.Shaw, Proc. Int. Joint Conf. on Artificial Intelligence, 1895.

[SIGKME1 87] **Proc. of First Workshop for the Special Interest Group on Knowledge Manipulation Engines,** Univ. of Reading, Jan. 1987.

[Sowa 84] **Conceptual Structures: Information Processing in Mind and Machine,** J.Sowa, Addison-Wesley, 1984.

[Stanfill 86] **Parallel Free Text Search on the Connection Machine,** C.Stanfill, B.Kahle, Comm ACM, Dec 1986.

[Stolfo 87] **Initial Performance of the DADO2 Prototype,** S.Stolfo, IEEE Computer, Jan. 1987.

[Stone 87] **Parallel Query of Large Databases: A Case Study,** H.Stone, IEEE Computer, Oct 1987.

[Ueda 86] **Guarded Horn Clauses,** K.Ueda, PhD Thesis, Univ. of Tokyo, March 1986.

[Valduriez 86] **Evaluation of Recursive Queries using Join Indices,** P.Valduriez, H.Boral, Proc. First Int. Conf. on Expert Database Systems, Charleston, 1986.

[Waltz 87] **Applications of the Connection Machine,** D.Waltz, IEEE Computer, Jan 1987.

[Warren 84] **Efficient Prolog Memory Management for Flexible Control Strategies,** D.S.Warren, Proc. of Int. Symposium on Logic Languages, Atlantic City, Feb. 1984.

[Warren 88a] **The SRI Model for OR Parallel Execution of Prolog - Abstract design and Implementational Issues,** D.H.D.Warren. BCS/Occam Users Group Conf. on Parallel Architectures for A.I., London, Feb.1988.

[Warren 88b] **Implementation of Prolog,** D.H.D.Warren, Tutorial No.3, Int. Conf. on Logic Programming, Seattle, Washington, Aug.1988.

[Williams 87a] **The Design of a Prolog Database Machine,** M.Williams, Proc. Workshop of Special Interest Group in Knowledge Manipulation Engines, Brunel Univ, May 1897.

[Williams 87b] **Benchmarks for Prolog from a Database Viewpoint,** M.Williams, P.Massey, J.Crammond, Proc. Workshop of Special Interest Group in Knowledge Manipulation Engines, Brunel Univ, May 1987.

[Wise 86] **Prolog Multiprocessors,** M.Wise, Prentice Hall, 1986.

[Woods 85] **What's in a Link: Foundations for Semantic Networks,** W.Woods, in **Readings in Knowledge Representation,** Ed. Brachman, Morgan Kaufmann, 1985.

Appendix A

Lexical Conventions for the Representation of Logic

Logical expressions used in the thesis fall into two categories: those which refer to examples in a specific programming language and those which represent generalised logic programming concepts. The lexical conventions used for these examples reflects this division.

a) Specific Logic Programming Languages

Examples of several programming languages are used in the thesis. These include Prolog, the Pure Logic Language, BRAVE and PEPSys Prolog. Where examples are refer to specific languages the "accepted" syntax for that languages is used. In the case of Prolog this is Edinburgh Prolog as defined by Clocksin and Mellish [Clocksin 81]. For the Pure Logic Language the language syntax is given in Appendix B; other language syntax definitions are referenced at the appropriate point in the text of the thesis.

b) Generalised Logic Languages

The thesis presents several examples of generalised logic expressions which are not specific to any recognised programming languages. For these examples the convention has been adopted that the syntax used should be based on the commonly accepted first order logic representation; connectives are represented by "and" and "or", logical implication by "<-", predicate names and variables are given in lower case character strings and scoping limits defined by the use of brackets.

Appendix B

Pure Logic Language Syntax

B1. Introduction

The syntax definition presented here refers to Version 0.2, Issue A, of the Pure Logic Language. It is based on the formal description prepared by MacBrien [MacBrien 88a] and uses standard BNF notation.

B2. Pure Logic Language Definition

B2.1. Symbols and Delimiters

<digit> ::= 1|2|3|4|5|6|7|8|9|0

<letter> ::= a|b|c....|z

<symbol> ::= !|$|@|% ....

<quote> ::= "

<bra> ::= (

<ket> ::=)

<list bra> ::= [

<list ket> ::=]

<range> ::= ..

<exist quant> ::= some

B2.2. Identifiers and Numbers

<identifier> ::=<letter> [<letter>|<digit>|<underline>]

<unsigned> ::= [<digit>]

<integer> ::= (+|-) <unsigned>

<float> ::= <integer>.(<unsigned>)e<integer>

<number> ::= <integer>|<float>

<string> ::= <quote> [<digit>|<letter>|<symbol>]<quote>

<atom> ::= <identifier>|<number>|<string>

<arithmetic atom> ::= <identifier>|<number>

B2.3. Structures

```
<identifier list>        := <bra>[<identifier>]<ket>
<parameter list>         ::= <bra>[<atom> | <list>]<ket>
<range structure>        ::= <atom><range><atom>
<list element>           ::= <atom> | <list> | <range structure>
<list>                   ::= <list bra>[<list element>]<list ket>
```

B2.4. Predicates and Operators

```
<not connective>         ::= not | ~
<and connective>         ::= and | &
<or connective>          ::= or | |
<times operator>         ::= *
<plus operator>          ::= +
<power operator>         ::= ^
<square root operator>   ::= sqrt
<cons operator>          ::= ::
<equals predicate>       ::= =
<greater predicate>      ::= >
<in predicate>           ::= in
<list predicate>         ::= list
<logic b_conn>           ::= <and connective> | <or connective>
<arithmetic b_op>        ::= <times operator> | <plus operator> |
                                   <power operator>
<arithmetic u_op>        ::= <square root operator>
<relational predicate>   ::= <equals predicate> | <greater predicate>
```

B2.5. Expressions

```
<arithmetic exp>         ::= <arithmetic atom>
<arithmetic exp>         ::= <arithmetic u_op><arithmetic exp>
<arithmetic exp>         ::= <arithmetic exp><arithmetic b_op>
                                                 <arithmetic exp>
<arithmetic exp>         ::= <bra><arithmetic exp><ket>
<data element>           ::= <arithmetic exp> | <string> | <list>
<data exp>               ::= <data element>
<data exp>               ::= <data exp><cons operator><list>
<data exp>               ::= <bra><data exp><ket>
```

<predicate> ::= <identifier>
<predicate> ::= <data exp><relational pred><data exp>
<predicate> ::= <atom> | <list><in predicate><list>
<rule> ::= <identifier><parameter list>
<logic exp> ::= <predicate>
<logic exp> ::= <rule>
<logic exp> ::= <logic exp><logic b_conn><logic exp>
<logic exp> ::= <not connective><logic exp>
<logic exp> ::= <bra><logic exp><ket>
<logic exp> ::= <exist quant><identifier list><bra>
<logic exp><ket>

B2.6. Command Line Interface

<opsys call> ::= <times operator><string>
<rule definition> ::= define <identifier><identifier list> tobe
<logic exp>?
<display command> ::= display <identifier>
<list command> ::= list
<query> ::= <logic exp>?
<exit command> ::= exit
<help command> ::= help
<clear command> ::= clear
<parallel command> ::= parallel

Appendix C

PLL Programs Used for Benchmark Testing

C1. Program 1 - Family Database

```
define married(x y) tobe spouse(x y) or spouse(y x)?
define stepparent(x y) tobe some(z)(married(z x) and parent(z y)  and
      not(parent(x y)))?
define grandparent(x y) tobe some(z)(parent(x z) and parent(z y))?
define sibling(x y) tobe some(z)(parent(z x) and parent(z y) and  not(x=y))?
define firstcousin(x y) tobe some(z)(grandparent(z x) and  grandparent(z y)
      and not(x=y) and not(sibling(x y)))?
define aunt(x y) tobe some(z)(female(x) and sibling(x z) and  parent(z y))?
define parent(x y) tobe ([x y] in [["fred" "bill"]["fred"  "ben"]
      ["fred"  "betty"]["fanny"  "bill"]["fanny"  "ben"]["fanny"  "betty"]
      ["bruce"  "scotty"]["bruce"  "simon"]["butch"  "sonia"]
      ["butch"  "sarah"]["bill"  "sue"]["bill"  "sam"]["babs"   "sue"]
      ["babs"  "sam"]["becky"  "sally"]["becky"  "seth"]["ben"   "seth"]
      ["ben"  "sally"]["betty"  "sarah"]["betty"  "sonia"]["betty"  "scotty"]
      ["betty"  "simon"]])?

define male(x) tobe ([x] in [["fred"]["bill"]
      ["ben"]["sam"]["seth"]["simon"]["scotty"]])?

define female(x) tobe ([x] in [["fanny"]["betty"]
      ["sarah"]["sally"]["sonia"]["sue"]])?

define spouse(x y) tobe ([x y] in [["fred" "fanny"]["bill" "babs"]
      ["ben"  "becky"]["betty"  "butch"]["betty"  "bruce"]])?
```

C2. Program 2 - Map Colouring and Other Sample Definitions

```
define a(x) tobe (x=99) and b(x)?
define b(x) tobe c(x) or d(x)?
define m(x) tobe n(x) and o(x)?
define n(x) tobe p(x) or q(x)?
define o(x) tobe r(x) or s(x)?
define smallest(a b) tobe (a in b) and not(some(c)((c in b) and  (a>c)))?
define div(x y) tobe (some(z)( (z in [0..y]) and (y=(z*x)) ) )?
```

```
define fact(x y) tobe ( (x=0) and (y=1)) or (some(a b)( (x in  [1..y]) and (x=(a+1))
      and (y=(x*b)) and fact(a b)))?

define append(a b c) tobe ((a=[]) and (b=c))
      or (some(x y z)
      ( (a=x::y)
      & (c=x::z)
      & append(y b z)))?

define ins(x y z) tobe (x=(y::z)) or
      (some(xhead xtail ztail)
      ( (x=(xhead::xtail))
      &  (z=(xhead::ztail))
      & ins(xtail y ztail)
      ))?

define perm(x y) tobe ( (x=[]) and (y=[]))
      or (some(xhead xtail remy)
      ( (x=(xhead::xtail))
      & (ins(y xhead remy))
      & (perm(xtail remy))
      ))?

define colour(a b c d e) tobe next(a b) and next(c d) and next(a c) and
      next(a d) and next(b c) and next(b e) and next(c e) and  next(d e)?

define next(a b) tobe [a b] in [["red" "blue"]["blue"  "red"]
      ["yellow" "red"]["green" "red"]["red" "yellow"]
      ["blue" "yellow"]["yellow" "blue"]["green" "blue"]
      ["red" "green"]["blue" "green"]["yellow" "green"]["green" "yellow"]]?
```

Appendix D

Analysis of Potential AND Parallelism in PLL Programs

The following rules (Fig.D1) are part of a test program developed by ICL to demonstrate the use of the Pure Logic Language. The top level rule "route" defines the set of possible routes between two nodes, and the node connections are stored in the expression "link". The linking rule is defined as an "in" expression and represents a number of alternatives, thus providing scope for OR parallelism. However for the purpose of this analysis OR parallelism is ignored and attention focused on AND parallel execution.

```
define route(x y r) tobe route2(x y r [])?
define route2(x y r visited) tobe
      ((x=y) and (r=[]))
   or  (not(x=y)
     & (some(mid rem visitmid)
     (link(x mid)
     & not (mid in visited)
     & (visitmid=mid::visited)
     & route2(mid y rem visitmid)
     & r=mid::rem
     )))?

define link(a b) tobe ([a b] in [ [1 2] [2 5] [3 2] [5 4]
 [4 3] [5 3] [3 6] [6 7] [7 8] ])?
```

Fig. D1 - Code for "route" rules

The expression tree for the query "route(x y r)" with x and y instantiated to appropriate variables is shown in Fig.D2. The intention is to show how the relationship between shared variables affects the possible concurrent execution of the query; the figures above each subexpression indicate the sequencing of their execution.

No OR parallelism is assumed, ie the alternative conjoined expressions derived from the first rewrite of "route(2 4 r)" are handled sequentially. However no alternatives are evaluated at Stage 7 as the analysis permits the successful "link" instantiations {x/5,y/4} to be realised at the first attempt.

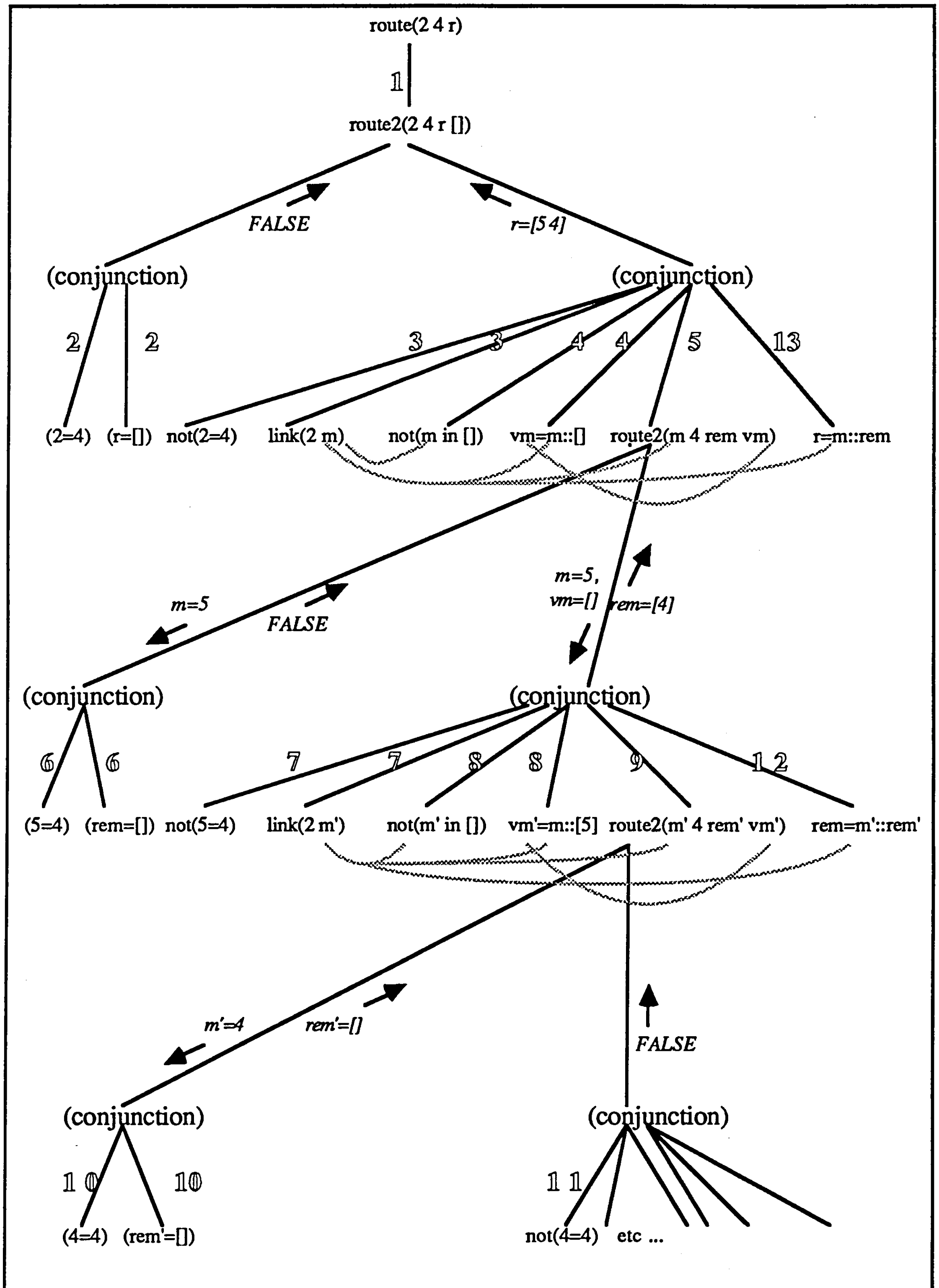


Fig. D2 - Expression Tree for Query route(2 4 r)?

If it is assumed that each subexpression can be evaluated in one unit of time, ie all processes take the same length of time to complete, it can be seen that for this small query there are 20 subexpressions to evaluate; by employing AND parallel execution this can be reduced to 13 steps because Stages 2, 3, 4, 6, 7, 8 and 10 allow for expression evaluation to be performed in parallel. The theoretical maximum potential speedup for this query is therefore 20/13, ie 1.54. This compares unfavourably with the maximum theoretical benefits obtainable from OR parallel execution for the same type of query.

Appendix E

OR Tree from PLL Benchmark Program

In order to show the detailed operation of OR parallel execution for one of the PLL benchmark programs the query:
aunt(x y)?
is considered. In the tests used this query generated 711 independent processes, 33 of which gave rise to spawning operations. Of the remainder, 8 produced binding values and 670 responded FALSE. However this number of processes makes detailed consideration of the execution paths difficult and for the purpose of this appendix, a "reduced" rule set is used. This involves a smaller number of "base predicate" instances in the rules for "parent" and "female" as shown in Fig.E1.

```
define aunt(x y) tobe
     (some(z)(female(x) and sibling(x z) and parent(z y)))?

define sibling(x y) tobe
     (some(z)(parent(z x) and parent(z y) and not(x=y)))?

define female(x) tobe ([x] in [["fanny"] ["betty"] ["sue"]])?

define parent(x y) tobe ([x y] in [["fred" "betty"] ["fred" "ben"]
     ["bill" "sue"] ["ben" "seth"] ["fanny" "betty"]])?
```

Fig. E1 - "Reduced" Rule Base

The first rewrite results in the conjunction:
some(z)(female(x) and sibling(x z) and parent(z y)?
The next rewrite leads to the spawning of processes when female(x) is split into its different branches. These three first level OR processes represent the expression:
sibling(x c) and parent(x z))?
with x instantiated to "fanny", "betty" and "sue" respectively.

When the first OR process (1.1) executes the expression is rewritten into:
some(w)(parent(w x) and parent(w z) and not(x=z) and parent(z y))?
which leads to further process spawning when parent(w x) is rewritten. Five processes are created, all of which fail on subsequent rewrites because no suitable "parent" match is found with x instantiated to "fanny". This is

shown diagrammatically in Fig.E2. where all leaf node processes are assumed to respond FALSE unless otherwise marked.

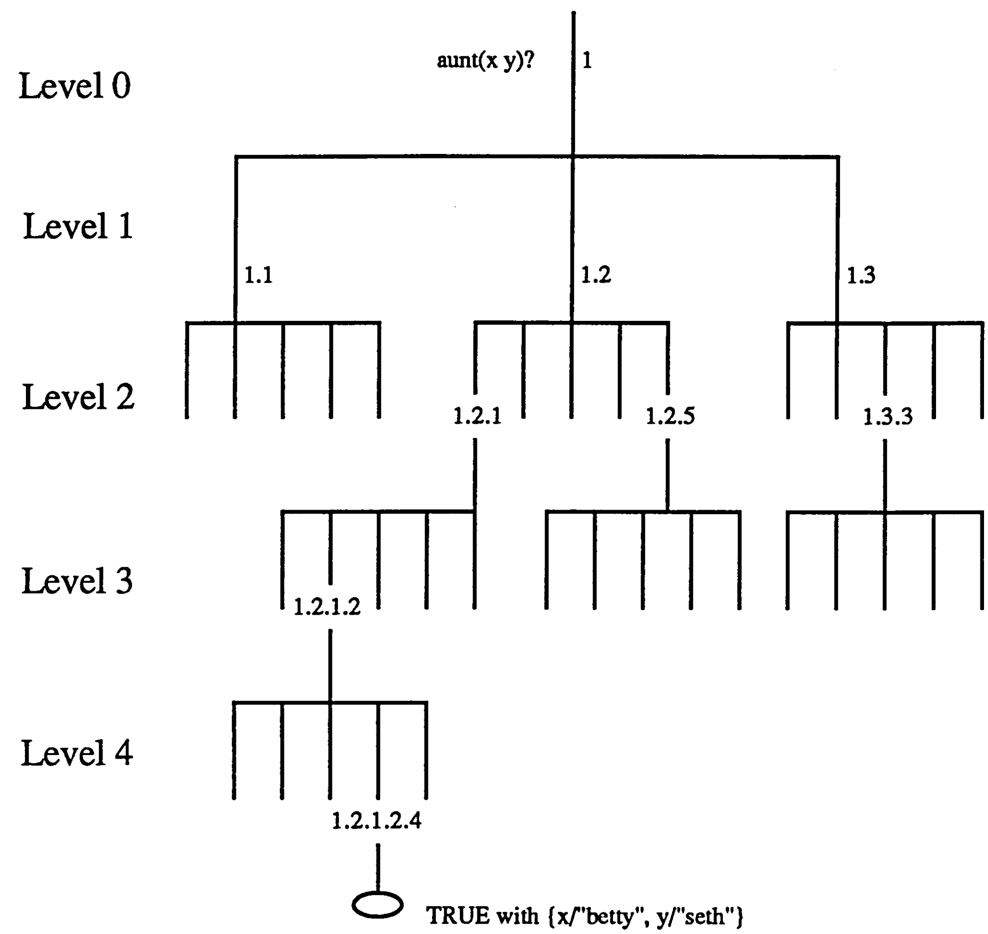


Fig. E2 - Solution Tree

The second OR process (1.2) is similarly transformed but in this instance two of the Level 2 OR processes succeed to produce binding values with x = "betty". Thus Processes 1.2.1 and 1.2.5 represent the expression:
parent(w z) and not(x=z) and parent(z y)?
with Process 1.2.1 having binding values {x/"betty", w/"fred"}, and Process 1.2.5 {x/"betty", w/"fanny"}. In similar fashion Process 1.3. spawns five processes of which Process 1.3.3 results in further processes. Thus the original three Level 1 processes have produced fifteen Level 2 and fifteen Level 3 processes as shown. Only one Level 3 process does not end in failure: this is Process 1.2.1.2 representing the expression:
parent(z y)?
with bindings {x/"betty",w/"fred",z/"ben"}.

Five final Level 4 processes are spawned and Process 1.2.1.2.4. succeeds to bind y to "seth", thus returning {x/"betty",y/"seth"} to the user.

If it is assumed that each process has the same execution time the theoretical maximum number of processes that can run concurrently is fifteen and this parallelism will occur at Levels 2 and 3 during query evaluation. The mean amount of parallelism for the period of query evaluation is the total number of processes divided by the number of levels of spawning, ie (1+3+15+15+5)/5 = 7.8. However this hypothetical approach is not likely to produce accurate predictions for the parallel PLL system as currently implemented because processes vary considerably in their execution time, but it indicates that OR parallelism does give rise to the potential for performance benefits. The full version of the query "aunt(x y)" produces 711 processes within the same five levels of spawning, and thus with a larger set of base predicates the potential for OR parallelism increases considerably.

Appendix F.

The Parallel Simulation Software

F1. Introduction

The software written specifically for this project can be divided into three components: the small PLL programs used to test the system, the data interpretation program which was used to process the results of the simulation, and the Parallel PLL simulation system itself. This appendix is concerned with the simulation software. The PLL programs are given in Appendices C and H, and the examples of the output of the data interpretation program can be seen in Appendix G.

The parallel simulation program consists of several modules written in C. These are:

pll_main.c	- the main program,
pll_parser.c	- the software responsible for parsing an incoming query into an expression tree,
pll_memory.c	- the general memory management functions,
pll_core.c	- the sequential rule rewrite manager,
pll_par_core.c	- the parallel rule rewrite manager,
pll_parallel.c	- the parallel machine emulation module,
pll_maths.c	- library of mathematical functions,
pll_lists.c.	- library of list processing functions.

The two modules that have been written during this project are: pll_par_core.c and pll_parallel.c. The source code for the whole system occupies over 200 Kbytes, pll_parallel.c and pll_par_core.c representing approximately 70 Kbytes and 40 Kbytes respectively.

F2. The Parallel Machine Emulation Module.

F2.1. Introduction

The parallel machine emulation functions are called from the main program (pll_main.c) after the incoming query has been parsed. The top level function <parallel_machine_driver> is responsible for the control of the machine simulation and it relates to the other high level functions as shown in Fig.F1.

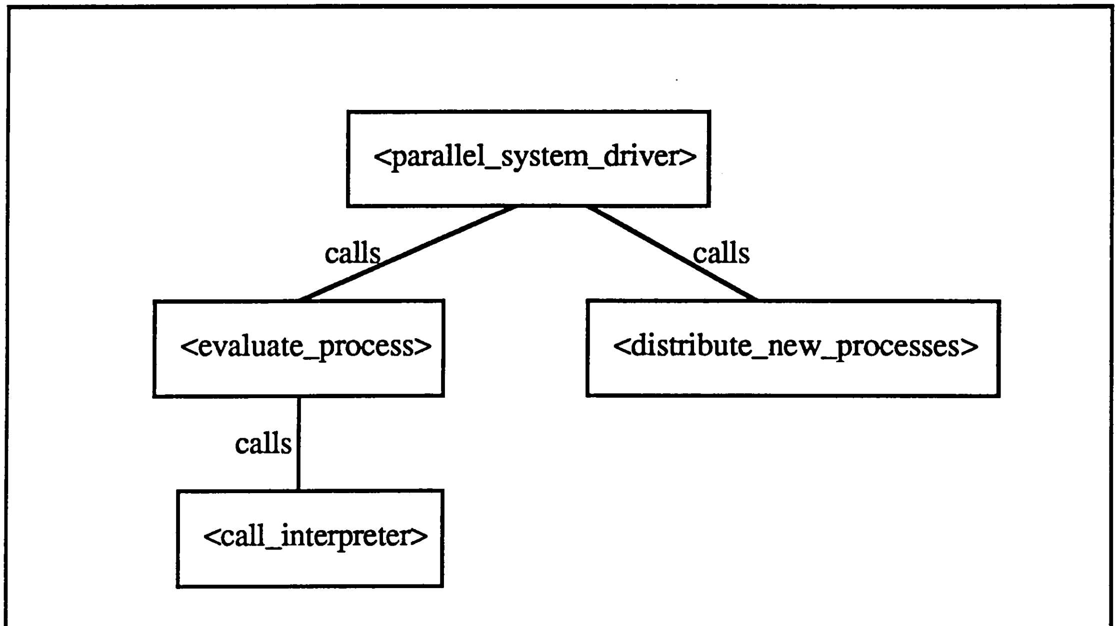


Fig. F1 - Top Level Functions

The following sections contain outline details of the main data structures used in this module as well as the description and code of the high level functions and brief details of the different groups of lower level functions.

F2.2. Data Structures.

F2.2.1. Machine Emulation Structures

Two main structures are defined for the machine; a controller and an array of pes (processing elements). These represent the physical machine. Each pe has two local queues: one for processes awaiting allocation/ distribution, and the other to hold processes for execution, plus some temporary storage used during process spawning operations. Data on the usage patterns of the input memories is also stored in each pe. The controller holds information on the number of processes awaiting execution and the finish time of the last process to run in each pe: this is used to ensure that processes get allocated to the least busy pes. It also holds information on bus availability.

```
typedef struct
     {int *execution_queue;
      int *allocation_queue;
      int *temp_queue;
      int *temp_bindings_list;
      int max_memory_value[MAX_NO_BUSSES];
      int time_max_value[MAX_NO_BUSSES];
     }
 PE_TYPE;

typedef struct
     {int state_of_pe[MAX_NO_PES];
      int process_finish_pe[MAX_NO_PES];
      int bus_finish_time[MAX_NO_BUSSES];
     }
 CONTROLLER_TYPE;
```

F2.2.2. Process Representation

These have been detailed in Chapters 5.4.3 and 7.4.2. There are three main structures used to represent the parallel processes within the system: processes structures which include processes descriptions, process records and allocation records. Process and allocation records are held on the global control queues, ready_to_run_queue and read_to_allocate_queue, to allow easy access to the processes awaiting action. Processes structures are held on the local queues within processing elements and represent the actual processes defined by the parallel rewrite interpreter.

F2.3. Machine Emulation Functions

F2.3.1. Function: Parallel_System_Driver

The code for this function is shown in Fig.F2. This top level function is responsible for configuring and initialising the parallel machine and for "driving" the software that emulates the interpreter running in the parallel system. It steps through the execution checking that all necessary processing has been completed before incrementing the timing point. Processes within each timestep are executed or distributed as appropriate.

F2.3.2. Function: Evaluate_Process

This function (Fig.F3) is called from <parallel_system_driver>. It is responsible for the administrative tasks performed each time a process is executed. Process evaluation is accomplished by the call to <call_interpreter> within this function.

F2.3.3. Function: Call_Interpreter

This function is shown in Fig.F4; it converts the process into an expression tree and then passes control to the parallel rule rewrite module by the call to <rewrite_expP>. It is also responsible for inserting timing functions for the three main tasks of the function.

F2.3.4. Function: Distribute_New_Processes

This function (Fig.F5) controls the distribution of spawned processes throughout the machine. It performs this task subject to two constraints: it has to confirm that there are processes awaiting allocation within the timing limits, and also that a bus is available within the same time interval. Having checked these conditions the function <distribute_allocation_record> is called and this implements the allocation of processes to processing elements and performs their transfer. The time taken by the broadcast is calculated and the appropriate data stores are updated accordingly.

F2.3.5. Low Level Functions

F2.3.5.1. Basic Functions

These functions define the nodes used for process_records, allocation records, and processes. Garbage collection functions are also defined.

```
int *node4(a, b, c, d)
int *node5(a, b, c, d, e)
void release_node4(p)
void release_node5(p)
```

```
/* Top level function which drives the parallel simulation. It "timesteps" through the
query evaluation executing and distributing processes as appropriate */


void parallel_system_driver(exp)
int *exp;
{int process_finish_time;
 int query_time=0;
 int timing_point=TIMESTEP;
 int processing_complete=FALSE;
 int processing_within_timestep=TRUE;
 int *temp; int n; int i,j;
 initialise_machine();
 set_up_query(exp);
 while (!processing_complete)
   {while (processing_within_timestep)
     {temp=ready_to_run_queue;
     /* Check each record on the ready_to_run_queue and evaluate process if it falls within the timestep */
     while (temp!=NULL)
      {n=temp[PE];
      if ((temp[TIME]<=timing_point) && (controller.process_finish_pe[n]<=timing_point))
        {process_finish_time=evaluate_process(temp);
        if (process_finish_time > query_time)
         {query_time=process_finish_time;
         }
        }
      temp=temp[NEXT_PROC];
     }
    /* All possible processes have been executed */
   processing_within_timestep=FALSE;

 /* Now check ready_to_allocate_queue. If there are processes to allocate
 distribute_new_processes will attempt to do this. It will return a TRUE value if distribution
 has been successful and new processes have been placed on the ready_to_run_queue */

 if (ready_to_allocate_queue)
      {processing_within_timestep=distribute_new_processes(timing_point);
     }
   }
   timing_point+=TIMESTEP;
   processing_within_timestep=TRUE;
   if ((ready_to_run_queue==NULL)&&(ready_to_allocate_queue==NULL))
     {processing_complete=TRUE;
     }
  }
 printf("ALL DONE! Time taken = %d microsecs\n", query_time);
 printf("No of processes = %d\n", proc_no);
/* Now output stored timing data */
 timing_data_to_file();
/* Now output results data */
 results_data_to_file();
}
```

Fig. F2 - Code for <parallel_system_driver>

```
/* High level function which initiates and controls process evaluation */

int evaluate_process(proc_record)
int *proc_record;
{int resulting_packet_size; /* the interpreter will return a positive
                       value if a data packet has been formed */
 int process_finish_time;
 int n=proc_record[PE];
 int *process=pe[n].execution_queue;
 int *temp;
 ready_to_run_queue=remove_from_queue(proc_record,ready_to_run_queue);
 while (process[PROC_NO] != proc_record[PROC_NO])
  {process=process[NEXT_PROC];
  }
 /* Release old process_record node*/
 release_node4(proc_record);
 /*Now check to see if process has been subjected to a delay while on the execution queue */
 if (controller.process_finish_pe[n]>process[TIME])
   {process[TIME]=controller.process_finish_pe[n];
   }
 /*check on contents of the input memories and make suitable updates*/
 update_input_memories(n,process);
 if (execution_trace)
  {printf("Evaluate_process executing process no.%d with start time %d on
            pe no.%d\n",process[PROC_NO],process[TIME],n);
  }
 resulting_packet_size=call_interpreter(process,n);
 pe[n].execution_queue=remove_from_queue(process,pe[n].execution_queue);
 process_finish_time=process_timing_queue[TIME] +process_timing_queue[SET_UP_TIME]
                 +process_timing_queue[EVAL_TIME]+process_timing_queue[SPAWN_TIME];
 controller.process_finish_pe[n]=process_finish_time;
 process_timing_queue[PROC_NO]=NULL;
 process_timing_queue[TIME]=NULL;
 process_timing_queue[SET_UP_TIME]=NULL;
 process_timing_queue[EVAL_TIME]=NULL;
 process_timing_queue[SPAWN_TIME]=NULL;
 controller.state_of_pe[n]--;
 if (resulting_packet_size!=0)
  {if (execution_trace)
   {printf("Process no.%d finished, processes spawned\n\n",process[PROC_NO]);
   }
  ready_to_allocate_queue=append_to_queue(create_allocation_record
                                   (n,resulting_packet_size), ready_to_allocate_queue);
  pe[n].allocation_queue=join_queues(pe[n].allocation_queue, pe[n].temp_queue);
  pe[n].temp_queue=NULL;
  }
 else
 {if (execution_trace)
  {printf("Process no.%d finished, no processes spawned\n",process[PROC_NO]);
  }
 }
 release_node5(process);
 return(process_finish_time);
}
```

Fig. F3 - Code for <evaluate_process>

```
/* This function creates an expression tree and invokes the rewrite manager */
int call_interpreter(process,n)
int n; int *process;
{int timer_now();
 int *proc_desc=process[PROC_DESC];
 int *exp, *pter,*r;
 int resulting_packet_size, results_size, add_to_spawn_time;
 int combine_and_or_time=0;
 int time1, time2, set_up, eval, spawn, debind;
 start_spawn_time=0; finish_spawn_time=0;
 process_timing_queue[PROC_NO]=process[PROC_NO];
 process_timing_queue[TIME]=process[TIME];
 time1=timer_now();
 exp=convert_process_desc(proc_desc,n);
 time2=timer_now();
 set_up=(time2-time1);
 process_timing_queue[SET_UP_TIME]=set_up;
 time1=timer_now();
 pter=rewrite_expP(exp,n);
 time2=timer_now();
 add_to_spawn_time=(finish_spawn_time-start_spawn_time);
 eval=((time2-time1)-add_to_spawn_time);
 process_timing_queue[EVAL_TIME]=eval;
 if (pter[NAME]==AND_OR)
    {time1=timer_now();
     pe[n].temp_queue=combine_and_or(pter,pe[n].temp_queue);
     time2=timer_now();
     combine_and_or_time=(time2-time1);
    }
 if ((pter[NAME]==OR) || (pter[NAME]==AND_OR))
    {pe[n].temp_queue=copy_spawn_process(pe[n].temp_queue);
     time1=timer_now();
     add_bindings_to_process_desc(pe[n].temp_queue,TRUE);
     time2=timer_now();
     spawn=((time2-time1)+combine_and_or_time+add_to_spawn_time);
     resulting_packet_size=calculate_process_packet(pe[n].temp_queue);
     process_timing_queue[SPAWN_TIME]=spawn;
     time_stamp_processes(pe[n].temp_queue, (process[TIME]+set_up+eval+spawn));
     if (execution_trace)
       {printf("Measured process time = %d. ",(set_up+eval+spawn));
        printf("Elapsed time since start of query = %d\n",pe[n].temp_queue[TIME]);
       }
     output_timings_to_file(process_timing_queue, process_function_calls,n);
     return(resulting_packet_size);
    }
 else
   {if (pter!=FALSE_NODE)
      {int *results_process;
       int *results_process_desc=create_process_desc(pter);
       results_process=node5(NULL,NULL,NULL,NULL,results_process_desc);
       add_bindings_to_process_desc(results_process,FALSE);
       results_size=get_process_size(results_process,TRUE);
       printf("Results packet size= %d\n",results_size);
      }
    time1=timer_now();
    r=join_and(pter,debind_vars(0));
    time2=timer_now();
    debind=(time2-time1);
    process_timing_queue[EVAL_TIME]+=debind;
    if (execution_trace)
      {printf("Interpreter has returned expression:\n");
       print_exp(r); printf("\n"); printf("Measured process time = %d. ",(set_up+eval+debind));
       printf("Elapsed time since start of query =%d\n",process[TIME]+set_up+eval+debind);
      }
    else
     {if (r!=FALSE_NODE)
       {print_exp(r); printf("\n"); printf("Result from proc.no.%d\n",process[PROC_NO]);
       }
      }
    output_timings_to_file(process_timing_queue,process_function_calls,n);
    release_exp(r);
    return(0);   /* No packet for distribution */
}}}
```

Fig. F4 - Code for <call_interpreter>

F2.3.5.2. Queue Manipulation Functions

This section holds the general queue manipulation functions. These functions can be used for both four node and five node queues because the linking pointer is the fourth field in both cases.

```
int *append_to_queue(p, queue)
void check_queue(queue)
int *remove_from_queue(p, queue)
int *join_queues(queue1, queue2)
```

F2.3.5.3. Process Creation and Manipulation Functions

These functions are responsible for the basic operations that take place on processes, process_records and allocation records. They utilise some of the basic queue manipulation functions. Related to them are the functions for the spawning of new processes. These are called from the rule rewrite manager but are defined here because they also utilise the queue manipulation functions.

```
void create_process_desc1(node,list_pter)
int *create_process_desc(p)
void print_node_name(node)
int check_pointer_location(p)
void check_full_process_desc(proc_desc)
int *construct_bindings()
void add_bindings_to_process_desc(process,reset_stack)
int *create_process(proc_desc)
void release_process(p)
int create_process_record(p,n)
int *create_and_tree(process_desc,n)
void reinstate_bindings(binding_pter)
int *convert_process_desc(proc_desc,n)
int *combine_and_or(p,queue)
int *spawn_or_process(p,queue)
int *copy_spawn_process(queue)
```

```
/* This function repeatedly chooses the "oldest" record within the timestep
from the ready_to_allocate_queue and passes it to the inner
function <distribute_allocation_record()> */

int distribute_new_processes(time)
int time;
{int bus_no;
 int transfer_start_time;
 int transfer_length;
 int allocation_possible=TRUE;
 int *record_to_allocate;
 if ((ready_to_allocate_queue==NULL)&&(execution_trace))
  {printf("No processes awaiting allocation\n");
   allocation_possible=FALSE;
  }
 else
  {while ((ready_to_allocate_queue) && (allocation_possible))
    {record_to_allocate=choose_earliest_record(ready_to_allocate_queue);
     if (record_to_allocate[TIME]<=time)
       {bus_no=allocate_bus_for_transfer(record_to_allocate[TIME],time);
        if (bus_no !=(-1))
         {if (controller.bus_finish_time[bus_no]<record_to_allocate[TIME])
           {transfer_start_time=record_to_allocate[TIME];
            if (execution_trace)
             {printf("No delay in bus transfer\n");
             }
           }
          else
           {transfer_start_time=(controller.bus_finish_time[bus_no]+1);
           }
          transfer_length=distribute_allocation_record(record_to_allocate[TIME],
                               record_to_allocate[PROC_NO],record_to_allocate[PE],
                               record_to_allocate[PACKET_SIZE], transfer_start_time,bus_no);
          controller.bus_finish_time[bus_no]=(transfer_start_time+transfer_length);
          ready_to_allocate_queue=remove_from_queue(record_to_allocate,ready_to_allocate_queue);
          release_node5(record_to_allocate);
         }
       else
        {if (execution_trace)
          {printf("No bus available at present\n");
          }
         allocation_possible=FALSE;
        }
       }
     else
      {if (execution_trace)
        {printf("No suitable record to allocate\n");
        }
       allocation_possible=FALSE;
      }
    }
  }
 return(allocation_possible);
}
```

Fig. F5 - Code for <distribute_new_processes>

F2.3.5.4. Packet Communication Calculation Functions

These functions are used to calculate the size of the data packets used to implement the transfer of data throughout the machine.

```
int *calculate_variable_nos(paralist, var_nos, temp_array)
int *calculate_expression_vars(node, no_vars, temp_array)
void calculate_list_bindings(node,bindings)
int calculate_process_packet(process)
```

F2.3.5.5. Timing Functions

This group of functions are used to store data on timings during program execution.

```
void time_stamp_processes(process_queue, time)
void output_timings_to_file(timing_record,function_calls,pe_no)
void output_bus_data_to_file
    (bus_no,pe_no,packet_size,time_of_transfer,transfer_length,
                                                no_of_ procs,delay)
void timing_data_to_file()
void results_data_to_file()
```

F2.3.5.6. Memory Management and Checking Functions

The "size" of an individual process is calculated to give information on results packet size; for this data packet quantified variables are ignored as they are not returned to the user.

```
int get_process_size(process,ignore_quantified)
int check_node_visited(node_list)
void update_input_memories(n,process)
```

F2.3.5.7. Machine Configuration and Initialisation Functions.

The number of processing elements and busses in the parallel machine can be specified by the user at the start of each session.

void explain_cline()
void configure_machine()
void initialise_machine()
void set_up_query(p)

F2.3.5.8. Allocation and Distribution Functions

These functions implement the broadcasting of data packets in the machine. Processing elements and busses are allocated by reference to the data on usage stored in the controller.

int allocate_pe_for_process(time)
int *create_allocation_record(n, size)
int choose_earliest_record(record)
int allocate_bus_for_transfer(allocation_time,present_time)
void update_state_of_pes(transfer_time)
int distribute_allocation_record
(time,no_of_procs,n,packet_size,time_of_transfer,bus_no)

F3. The Parallel Rewrite Manager Module

F3.1. Introduction

The top level function in this module is <rewrite_expP> which is called from parallel machine emulation module by the function <call_interpreter>. It implements the rewriting of expressions by identifying the node at the root of the expression tree and passing control to the appropriate rewrite rule. These rules are encapsulated in the "eval" functions which are defined for each node type. Rewriting of nodes involves operations on the appropriate expression tree and as such most of the "eval" functions are based on recursive algorithms.

The rewrite rules which are of fundamental importance to the parallel interpreter are those dealing with conjoined and disjoined expressions, ie <eval_andP>, <eval_orP> and <eval_inP>. The manner in which these functions relate to the rest of the module is shown in Fig.F6. Many of the "eval" functions in the parallel rewrite manager were implemented by making minor alterations to the corresponding functions in the sequential system. However the move to a parallel process based system which produced spawned OR processes meant that new rewrite rules were needed for any node representing alternatives, ie OR, IN, RANGE and for conjunction rewrite rule <eval_and>. Details of the operation of these functions has been discussed in the main body of the thesis (Chapter 5.4.4.) and the code is presented in the next section.

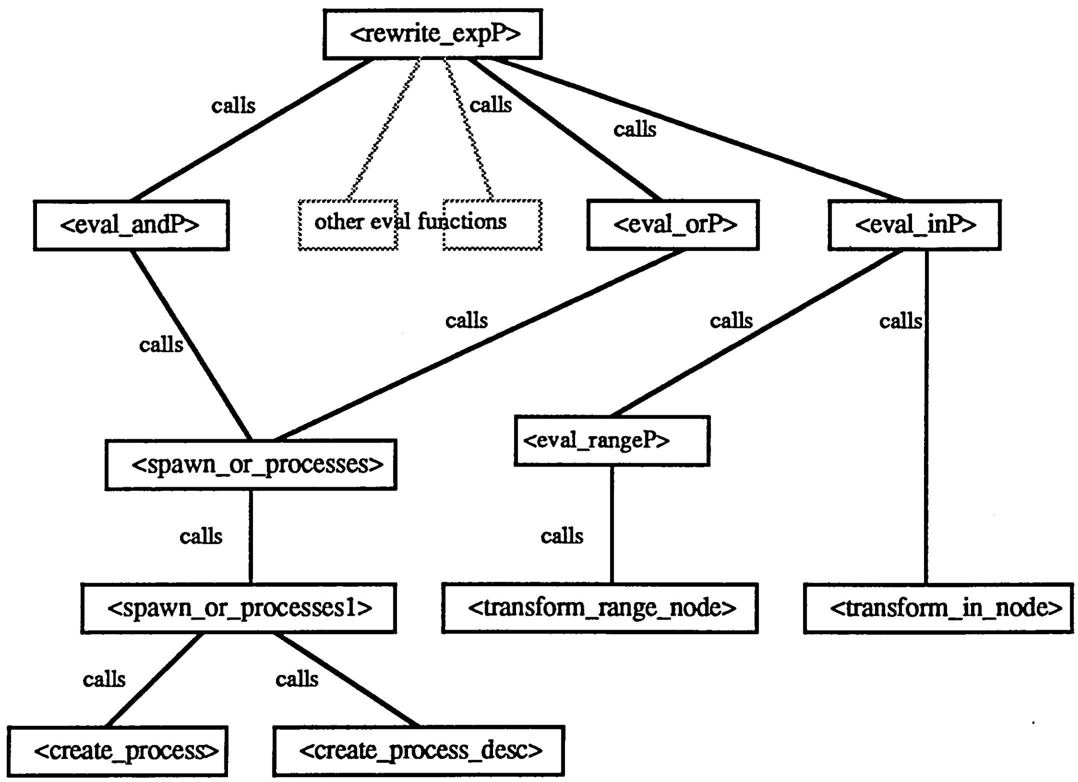


Fig. F6 - Function Calling in the Parallel Rewrite Manager

F3.2. Top Level Rewrite Function

This function operates to distinguish the node at the root of an expression tree and passes the tree to the appropriate rewrite rule as implemented in the "eval" functions. The code is given in Fig.F7.

F3.3. Node Rewrite or Eval Functions

F3.3.1. Conjunction Rewriting

The function <eval_andP> implements the algorithm which performs rewrites on conjoined expressions - see Fig.F8. The left hand node is rewritten and then the right hand one, and this process is repeated until no further bindings are made. Prior to each rewrite it checks to see if an alternative node has been encountered, in which case control is then passed to the process spawning functions.

```
/* This is the top level function in the rewriting module: it passes the expression tree
 to the appropriate "eval" function */

int *rewrite_expP(p,n)
int *p[]; int n;
{ int *eval_equalP(),*eval_andP(),*eval_orP(),
      *eval_gtP(),*eval_expP(),*eval_inP(),*eval_notP();

 switch (p[NAME])
 { case TERM    : return(TRUE_NODE);
   case NOT     : return(eval_notP(p,n));
   case EQUAL   : return(eval_equalP(p,n));
   case AND     : return(eval_andP(p,n));
   case GT      : return(eval_gtP(p,n));
   case IN      : return(rewrite_expP(eval_inP(p,n),n));
   case SOME    : return(rewrite_expP(p[RIGHT],n));
   case OR      : return(eval_orP(p,n));
   case CALL    : return(eval_ruleP(p,n));
   case EXP     : release_node2(p);
                  return(rewrite_expP(p[BODY]),n);
   default      : return(eval_expP(p,FALSE_NODE,n));
 }
}
```

Fig. F7 - Code for <rewrite_expP>

F2.3.2. Disjunction Rewriting

As shown in Fig.F6 there are several functions concerned with the evaluation of alternatives. These involve rewriting of OR, IN and RANGE nodes. The task of the primary function <eval_orP> is to pass the expression tree to the routine which implements the spawning of processes. This function <spawn_new_processes> is held in the parallel machine module as it involves the creation and manipulation of data structures defined for the simulation system. The other evaluation functions which deal with alternatives, ie <eval_inP> and <eval_rangeP>, transform the IN or RANGE node representation into a nested OR tree as discussed in Chapter 5.4.4 and this tree is then passed to <eval_orP>. In addition two "transformation" functions are defined to assist in the process of converting IN and RANGE nodes into nested OR trees. These latter functions are all held in the parallel rewrite manager module.

The code for the three "eval" functions is given below together with that for <spawn_new_processes> and the transformation operations (Figs.F9-F14).

```
/* The AND node rewriting function */

int *eval_andP(p,n)
int *p[]; int n;
{ int r_bl=-1;
 int l_bl;
 if (p[LEFT][NAME]==OR)
   {pe[n].temp_queue=spawn_or_process(p[LEFT],pe[n].temp_queue);
    return(node3(AND_OR,NULL,p[RIGHT]));
   }
 if (p[LEFT][NAME]==IN)
   {p[LEFT]=eval_inP(p[LEFT],n);
    if (p[LEFT][NAME]==OR)
     {pe[n].temp_queue=spawn_or_process(p[LEFT],pe[n].temp_queue);
      return(node3(AND_OR,NULL,p[RIGHT]));
     }
   }
if (p[RIGHT][NAME]==OR)
   {pe[n].temp_queue=spawn_or_process(p[RIGHT],pe[n].temp_queue);
    return(node3(AND_OR,p[LEFT],NULL));
   }
 if (p[RIGHT][NAME]==IN)
   {p[RIGHT]=eval_inP(p[RIGHT],n);
    if (p[RIGHT][NAME]==OR)
     {pe[n].temp_queue=spawn_or_process(p[RIGHT],pe[n].temp_queue);
      return(node3(AND_OR,p[LEFT],NULL));
     }
   }
 do
 { l_bl=binding_level;
  p[LEFT]=rewrite_expP(p[LEFT],n);
  switch(p[LEFT][NAME])
   {case AND_OR   : return(node3(AND_OR,p[LEFT],p[RIGHT]));
                    break;
    case OR         : return(node3(AND_OR,NULL,p[RIGHT]));
                    break;
    case TRUE       : release_node3(p);
                     return(rewrite_expP(p[RIGHT],n));
    case FALSE      : release_exp(p);
                     return(FALSE_NODE);
   }
  if (r_bl==binding_level)
   {break;
   }
  r_bl=binding_level;
  p[RIGHT]=rewrite_expP(p[RIGHT],n);
  switch(p[RIGHT][NAME])
   {case AND_OR  : return(node3(AND_OR,p[LEFT],p[RIGHT]));
                   break;
    case OR        : return(node3(AND_OR,p[LEFT],NULL));
                   break;
    case TRUE      : release_node3(p);
                   return(rewrite_expP(p[LEFT],n));
    case FALSE     : release_exp(p);
                    return(FALSE_NODE);
   }
  }
 while (l_bl!=binding_level);
 return(p);
}
```

Fig. F8 - Code for <eval_andP>

```
/* The OR node rewriting functions */

int *eval_orP(p,n)
int *p; int n;
 {pe[n].temp_queue=spawn_or_process(p,pe[n].temp_queue);
 return(p);
 }
```

Fig. F9 - Code for <eval_orP>

```
/* The high level spawning function which includes timing functions */

int *spawn_or_process(p,queue)
int *p; int *queue;
{start_spawn_time=timer_now();
 queue=spawn_or_process1(p,0);
 finish_spawn_time=timer_now();
 return(queue);
}



/* The inner spawning function which creates a queue of alternative processes
   when an OR node is found */

int *spawn_or_process1(p,queue)
int *p; int *queue;
{int *exp;
 int *temp;
 switch(p[NAME])
  {case  OR: { queue=spawn_or_process1(p[LEFT],queue);
               queue=spawn_or_process1(p[RIGHT],queue);
               break;
                 }
   default : {exp=create_process_desc(p);
               temp=create_process(exp);
               temp[NEXT_PROC]=queue;
               queue=temp;
               break;
              }
  }
 return(queue);
}
```

Fig. F10 - Code for Spawning Functions

```
/* The evaluation of IN nodes: if RANGE nodes are encountered separate function is callled */

int *eval_inP(p,n)
int *p[]; int n;
 {int *eval_rangeP();
 int *pter;
 switch(p[RIGHT][NAME])
  {case RANGE  : return(eval_rangeP(p,n));
   case IDENT   : p[RIGHT]=eval_expP(p[RIGHT],FALSE_NODE,n);
                  if (p[RIGHT][NAME]!=IDENT)
                   {return (eval_inP(p,n));
                   }
                  return(p);
   case LIST    : if (p[RIGHT][BODY]==NULL)/*ie the empty list*/
                   {release_node3(p);
                   return(FALSE_NODE);
                   }
                 return(transform_in_node(p[RIGHT][BODY],p[LEFT]));
   default      : abend(TM);
  }

}
```

Fig. F11 - Code for <eval_inP>

```
/* This function converts an IN node into a tree containing OR
   or RANGE nodes*/

int *transform_in_node(p,left_value)
int *p[]; int *left_value;
{int *pter;
 if (p[CDR])
  {if (p[CAR][NAME]==RANGE)
     {pter=node3(OR,(node3(IN,copy_exp(left_value),p[CAR])),
                              transform_in_node(p[CDR],left_value));
     return(pter);
     }

 pter=node3(OR,(node3(EQUAL,copy_exp(left_value),p[CAR])),
                              transform_in_node(p[CDR],left_value));
  return(pter);
  }
 if (p[CAR][NAME]==RANGE)
  {return(node3(IN,copy_exp(left_value),p[CAR]));
  }
 return(node3(EQUAL,left_value,p[CAR]));
}
```

Fig. F12 - Code for <transform_in_node>

```
/* The function which evaluates RANGE nodes and if appropriate converts them
   into OR trees */

int *eval_rangeP(p,n)
int *p[]; int n;
{ int* prange;      /* Ranging atom */
  int* plo;         /* Lower limit of range */
  int* phi;         /* Higher limit of range */
  int* eval_expP();
  int pter;

  if (p[RIGHT][NAME]!=RANGE)
    {return(p);
    }
  p[RIGHT][LEFT]=eval_expP(p[RIGHT][LEFT],FALSE_NODE,n);
  p[RIGHT][RIGHT]=eval_expP(p[RIGHT][RIGHT],FALSE_NODE,n);
  p[LEFT]=eval_expP(p[LEFT],FALSE_NODE,n);

  plo=p[RIGHT][LEFT];
  phi=p[RIGHT][RIGHT];
  prange=p[LEFT];

  switch(prange[NAME])
  {case IDENT :if (plo[NAME]==NUM && phi[NAME]==NUM)
                 {if (plo[BODY]>phi[BODY])
                   {return(FALSE_NODE);
                   }
                  pter=transform_range_node(plo[BODY],phi[BODY],prange[BODY]);
                  return(pter);
                 }
               break;
   case NUM  :if (phi[NAME]==NUM && plo[NAME]==NUM)
                {release_exp(p);
                 if (plo[BODY]>phi[BODY])
                  {return(FALSE_NODE);
                  }
                 return ((plo[BODY]<=
                          prange[BODY] && prange[BODY]<=phi[BODY]) ?
                                         TRUE_NODE : FALSE_NODE);
                }
              if (phi[NAME]==NUM)
               {if (prange[BODY]>phi[BODY])
                 {return(FALSE_NODE);
                 }
                return(node3(OR,node3(GT,prange,plo),
                        node3(EQUAL,copy_exp(prange),copy_exp(plo))));

               }
             if (plo[NAME]==NUM)
              {if (plo[BODY]>prange[BODY])
                {return(FALSE_NODE);
                }
               return(node3(OR,node3(GT,phi,prange),
                      node3(EQUAL,copy_exp(phi),copy_exp(prange))));
              }
             break;
   default  : break;
  }
 return(p);
}
```

Fig. F13 - Code for <eval_rangeP>

```
/* The inner function responsible for the transformation of certain RANGE nodes into OR trees */

int *transform_range_node(plo_value, phi_value, prange_value)
int plo_value; int phi_value; int prange_value;
{int *pter;
 int temp;
 temp=plo_value+1;
 if (plo_value != phi_value)
  {pter=node3(OR,(node3(EQUAL,node2(IDENT,prange_value),node2(NUM,plo_value))),
                        transform_range_node(temp,phi_value,prange_value));
   return(pter);
  }
 return(node3(EQUAL,node2(IDENT,prange_value),node2(NUM,plo_value)));
}
```

Fig. F14 - Code for <transform_range_node>

Appendix G

Test Results

G1. Introduction

This appendix gives details of various test results obtained from the simulation. Two versions of the Parallel PLL system have been used: these are referred to as the "original" and "optimised" versions. In the "optimised version" an allowance of 7 microsecs per function call has been subtracted from the execution times of each process in an attempt to obtain predictive data on the performance benefits to be gained by a recoding of the PLL interpreter. This is discussed fully in Chapter 9.3.

The results listed below fall into six categories:
a) a sample output from the data interpretation program showing the different forms in which the test data was presented for analysis,
b) total query evaluation times for repeated runs of several queries, using both versions of the Parallel PLL,
c) data on the execution times of individual processes within a query evaluation run,
d) details of function calling during process execution,
e) information on bus usage during query evaluation,
f) input memory utilisation data.
g) information on the pattern of results return during query evaluation,

With the exception of the data used to demonstrate the data interpretation program (Appendix G2), the test results are based on the querying of the PLL rule bases presented in Appendix C.

G2. Data Interpretation Program

The following pages show the manner in which the data interpretation program presented the test results. the first section shows details on the individual processes occurring during the evaluation of a query; this is followed by the number of processing elements in use, and details on return of results, input memory utilisation on patterns of bus usage.

For this example, in order to keep the program output small, the query used was
"stepparent(x y)?"
which was put to the "reduced" family database as documented in Appendix E.

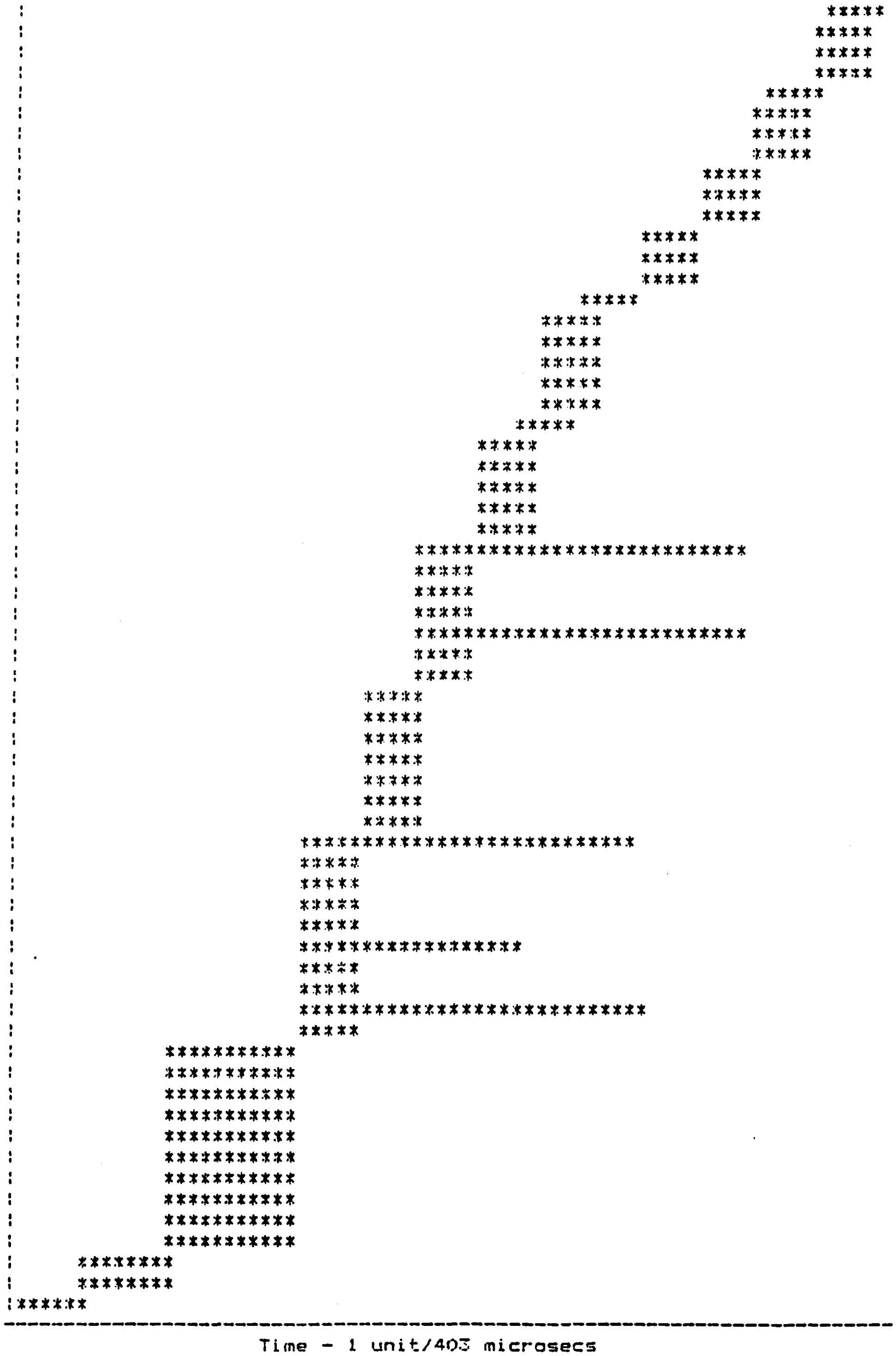
PROCESS TIMES

Proc_no	Set_up	Prcssing	Spawn	Total	%Pr_Crn
0	35	1654	463	2152	23
2	100	2015	919	3034	33
1	101	2018	914	3033	33
12	101	3172	947	4220	24
8	101	3172	940	4213	24
9	100	3175	947	4222	24
10	100	3175	946	4221	24
11	100	3176	948	4224	24
3	100	3174	946	4220	24
4	101	3174	946	4221	24
5	101	3173	946	4220	24
6	101	3175	948	4224	24
7	100	3173	947	4220	24
22	113	1845	0	1958	5
19	111	10934	0	11045	1
20	111	1821	0	1932	5
52	111	1813	0	1924	5
42	111	6991	0	7102	1
50	111	1814	0	1925	5
51	110	1823	0	1933	5
13	111	1816	0	1927	5
15	111	1824	0	1935	5
18	111	10711	0	10822	1
55	111	1837	0	1948	5
34	111	1824	0	1935	5
24	111	1824	0	1935	5
36	111	1836	0	1947	5
35	111	1827	0	1938	5
54	111	1822	0	1933	5
33	110	1821	0	1931	5
26	111	1925	0	1936	5
48	110	1819	0	1929	5
31	111	10686	0	10797	1
28	110	1823	0	1933	5
29	110	1818	0	1928	5
43	111	1819	0	1930	5
25	111	10682	0	10793	1
59	110	1820	0	1930	5
46	111	1821	0	1932	5
16	110	1830	0	1940	5
38	111	1822	0	1933	5
39	111	1823	0	1934	5
53	110	1821	0	1931	5
62	110	1811	0	1921	5
27	111	1810	0	1921	5
61	111	1818	0	1929	5
37	111	1812	0	1923	5
32	111	1810	0	1921	5
30	111	1826	0	1937	5
56	111	1828	0	1939	5
14	111	1814	0	1925	5
23	111	1823	0	1934	5
44	111	1818	0	1929	5
60	110	1818	0	1928	5
45	111	1825	0	1936	5
40	111	1813	0	1924	5
49	111	1814	0	1925	5
41	110	1815	0	1925	5
58	111	1816	0	1927	5
47	110	1810	0	1920	5
21	111	1816	0	1927	5
57	111	1811	0	1922	5
17	111	1812	0	1923	5

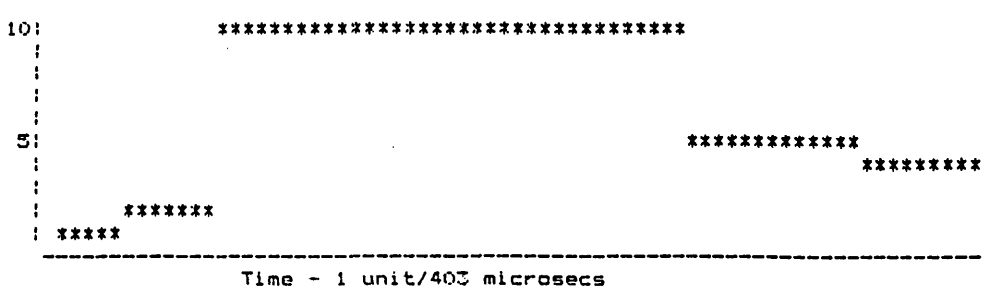
Times in microsecs

Total No Procs = 63; Spawning Procs = 13; Non Spawning Procs = 50
Average Set Up Time (all procs) = 107; Average Rewrite Time (all procs) = 2687
Average Rewrite Time (non spawning) = 2638
Average Set Up Time (non spawning) = 110
Average Rewrite Time (spawning) = 2878
Average Set Up Time (spawning)= 95
Average Spawn Time = 904

Output from Data Interpretation Program



PEs IN USE



Output from Data Interpretation Program

TIMES FOR RETURN OF RESULTS

Proc.No.	Time result found	Size of Packet	Pe.No.
18	20236	4	2
21	24078	4	4
25	24086	4	9

INPUT MEMORY DATA

Bus No.0 Max.No.Words	Bus No.1 Max.No.Words
32	8
32	8
8	24
16	16
0	32
8	24
16	16
16	16
32	0
24	8

Total time taken to complete query = 28173 microsecs

BUS TIMES

Bus No.	Sender PE	Start Time	Packet Size	No.Procs	Delay
0	0	2152	9	1	0
1	0	5186	9	4	0
0	1	5187	9	4	0
1	9	9404	8	4	0
0	0	9406	8	4	0
1	1	9409	8	4	2
0	3	9411	8	4	1
1	5	9414	8	4	4
0	4	9416	8	4	5
1	7	9419	8	4	7
0	8	9421	8	4	8
1	2	9424	8	4	10
0	6	9426	8	4	11

Times in microsecs

Output from Data Interpretation Program

G3. Total Query Evaluation Times

The following tables represent the total query evaluation times for the set of queries used with the rule bases given in Appendix C. Results are given for the original and optimised versions of the Parallel PLL.

QUERY - aunt(x y)? (original version)

No of PEs \ No of Busses	5	2	1
100	49987 48345 48535	50022 56521 56432	48367 49891 48225
50	65727 60186 70824	62055 68430 65649	64048 61928 64042
20	100254 109372 103519	103081 102373 100507	103000 104245 110524
10	176123 180789 177024	187114 194123 177861	180121 175978 181603
5	—	—	356360 340432 359381
3	—	—	562971 557888 562995
2	—	—	844017 843082 844012
Sequential	1773269		

Times in microsecs

Total Evaluation Times for query - aunt(x y)? (original version)

QUERY - grandparent(x y)? (original version)

No of PEs \ No of Busses	5	2	1
100	31902 31903 31890	31877 31890 31889	31883 31875 31885
50	40803 40810 40802	40816 40801 40811	40787 40798 40805
20	77159 76955 76941	76964 76950 77163	78740 78536 78709
10	130659 131120 131145	131029 132247 131038	132626 130954 131160
5	—	—	—
3	—	—	—
2	—	—	—
Sequential	870562		

Times in microsecs

Total Evaluation Times for query - grandparent(x y)? (original version)

QUERY - sibling(x y)? (original version)

No of PEs \ No of Busses	5	2	1
100	34058 33918 33994	34010 34307 34135	34210 34210 34225
50	44063 44076 44088	44155 44173 44332	44307 44593 44296
20	83967 83899 86001	85739 85852 84118	84198 84201 84188
10	145895 145675 145677	145737 145724 145544	145589 145464 146147
5	----	----	265865 265032 265772
3	----	----	429984 429977 429989
2	----	----	631621 630657 630623
Sequential	1255347		

Times in microsecs

Total Evaluation Times for query - sibling(x y)? (original version)

QUERY - firstcousin(x y)? (original version)

No of PEs \ No of Busses	5	2	1
100	526616 516641 487720	488990 668344 487527	587886 599514 480771
50	619727 550761 644894	659782 641927 641235	687037 559156 600270
20	1120775 1239646 1228172	1431747 1182410 1367141	1220644 1240654 1324455
10	1987514 2208643 2215851	2096960 2108650 2090921	2098536 2133817 2117696
5	—	—	—
3	—	—	—
2	—	—	9340797 9280050 9410059
Sequential	30327744		

Times in microsecs

Total Evaluation Times for query - firstcousin(x y)? (original version)

QUERY - stepparent(x y)? (original version)

No of PEs \ No of Busses	5	2	1
100	92710 92666 62210	92709 66481 68705	62263 80842 80861
50	100752 103173 98234	81815 88889 106330	98756 103987 93037
20	144515 140310 121342	129514 111744 116661	124368 147172 127145
10	222670 229555 193353	196755 219635 274120	181558 198193 199315
5	—	—	—
3	—	—	—
2	—	—	—
Sequential	Stack Full		

Times in microsecs

Total Evaluation Times for query - stepparent(x y)? (original version)

QUERY - colour("red" a b c d)? (original version)

No of PEs \ No of Busses	5	2	1
100	89993 84936 84787	85607 84977 92314	82283 82291 82860
50	137312 143154 142802	140559 140310 140205	138224 138768 135938
20	297157 299879 297897	297167 301567 303877	298989 295981 296920
10	560839 554005 559253	560582 555452 560273	553799 555823 563392
5	—	—	1090885 1091125 1091007
3	—	—	1808146 1803707 1806212
2	—	—	2706737 2706651 2707920
Sequential	5950949		

Times in microsecs

Total Evaluation Times for query - colour("red" a b c d)? (original version)

QUERY - aunt(x y)? (optimised version)

No of PEs \ No of Busses	5	2	1
100	45462 38865 43612	37890 46643 46966	39302 37806 38013
50	50320 45307 51449	48455 46844 45577	52547 51194 51190
20	82063 92244 80123	83331 80298 77668	77616 78927 80733
10	147563 150616 144694	150557 145234 148334	149444 146218 148668
5	—	—	—
3	—	—	—
2	—	—	—
Sequential	N/A		

Times in microsecs

Total Evaluation Times for query - aunt(x y)? (optimised version)

QUERY - grandparent(x y)? (optimised version)

No of Busses / No of PEs	5	2	1
100	24998 24987 24995	25088 25079 25076	25214 25202 25208
50	32268 32278 32270	32342 32351 32354	32487 32491 32485
20	62549 62762 62456	62691 62531 62686	62925 62797 62941
10	98888 98992 106172	99081 106205 98928	100425 98977 100430
5	---	---	---
3	---	---	---
2	---	---	---
Sequential	N/A		

Times in microsecs

Total Evaluation Times for query - grandparent(x y)? (optimised version)

QUERY - sibling(x y)? (optimised version)

No of PEs \ No of Busses	5	2	1
100	26858 26852 26877	26935 26916 26901	27378 27062 27061
50	35066 35039 35077	35200 35122 35217	35251 35282 35531
20	68602 68535 68866	68528 68627 68593	68501 68724 68474
10	110844 110398 110042	110021 110152 117180	110669 110668 110329
5	—	—	—
3	—	—	—
2	—	—	—
Sequential	N/A		

Times in microsecs

Total Evaluation Times for query - sibling(x y)? (optimised version)

QUERY - firstcousin(x y)? (optimised version)

No of Busses / No of PEs	5	2	1
100	387771 493498 599622	589561 434870 504384	498361 385503 485001
50	544928 428443 522160	509461 570234 507023	510055 618604 460311
20	1011282 962135 916753	1018924 979318 1001638	1035056 824611 1001635
10	1736799 1947863 1732450	1770767 1736870 1767492	1709457 1893068 1641310
5	—	—	—
3	—	—	—
2	—	—	—
Sequential	N/A		

Times in microsecs

Total Evaluation Times for query - firstcousin(x y)? (optimised version)

QUERY - stepparent(x y)? (optimised version)

No of PEs \ No of Busses	5	2	1
100	49920 61987 61872	49955 77303 76030	49829 50006 61468
50	70813 82343 71348	69019 92004 73570	53002 59918 84832
20	89881 89893 93176	138653 91324 125663	108227 90757 105992
10	168157 134605 132498	221240 133913 133915	125703 191439 164915
5	—	—	—
3	—	—	—
2	—	—	—
Sequential	N/A		

Times in microsecs

Total Evaluation Times for query - stepparent(x y)? (optimised version)

QUERY - colour("red" a b c d)? (optimised version)

No of PEs \ No of Busses	5	2	1
100	67717 69865 71352	71872 73493 69410	75636 68180 70123
50	109456 109473 119467	113754 113543 109590	115999 125120 119908
20	236073 237801 232378	236822 237802 242885	242678 237021 238115
10	446851 450499 461841	447214 451563 447128	449029 459115 456367
5	—	—	—
3	—	—	—
2	—	—	—
Sequential	N/A		

Times in microsecs

Total Evaluation Times for query - colour("red" a b c d)? (optimised version)

G4. Process Times

Details are given for the query
"aunt(x y)?"
of the times of individual processes. Processes are subdivided into three parts: set-up time, rewrite time and (where appropriate) spawn time.

PROCESS TIMES

Proc_no	Set_up	Prcssing	Spawn	Total	%Pr_Crn
0	35	2635	1076	3746	29
6	100	8909	3694	12703	29
1	101	8909	3688	12698	29
2	101	8910	3693	12704	29
3	100	8910	3693	12703	29
4	100	8910	3693	12703	29
5	100	8912	3693	12705	29
50	172	1785	0	1957	8
28	164	1761	0	1925	8
138	164	1763	0	1927	8
116	164	1763	0	1927	8
94	164	1761	0	1925	8
72	164	1755	0	1919	8
7	163	1752	0	1915	8
8	164	1754	0	1918	8
9	165	1753	0	1918	8
10	164	1754	0	1918	8
11	164	1756	0	1920	8
12	164	1757	0	1921	8
13	164	1761	0	1925	8
14	163	1762	0	1925	8
15	164	1763	0	1927	8
16	164	1763	0	1927	8
17	164	8416	3541	12121	30
18	172	1790	0	1962	8
19	164	8491	3602	12257	30
95	171	1787	0	1958	8
96	164	1757	0	1921	8
97	164	1759	0	1923	8
98	164	1756	0	1920	8
99	164	1759	0	1923	8
100	164	1758	0	1922	8
101	164	1767	0	1931	8
102	164	1766	0	1930	8
103	164	1765	0	1929	8
104	164	1772	0	1936	8
105	163	1765	0	1928	8
106	164	1772	0	1936	8
107	164	1766	0	1930	8
114	164	1763	0	1927	8
115	164	1761	0	1925	8
20	164	1763	0	1927	8
21	163	1763	0	1926	8
22	163	1762	0	1925	8
23	164	1762	0	1926	8
24	164	1762	0	1926	8
25	163	1761	0	1924	8
26	163	1762	0	1925	8
27	164	1762	0	1926	8
29	164	1755	0	1919	8
30	164	1753	0	1917	8
31	163	1755	0	1918	8
32	163	1755	0	1918	8
33	164	1753	0	1917	8
34	163	1755	0	1918	8
35	164	1755	0	1919	8
36	164	1755	0	1919	8
58	164	1757	0	1921	8
62	164	1759	0	1923	8
123	163	1767	0	1930	8
124	164	1766	0	1930	8
125	165	8414	3531	12110	30
126	171	1793	0	1964	8
127	165	1764	0	1929	8
128	164	1763	0	1927	8
129	164	1766	0	1930	8
130	163	1765	0	1928	8
59	164	1758	0	1922	8
60	164	1758	0	1922	8
61	164	1758	0	1922	8
65	164	1753	0	1917	8
66	164	1756	0	1920	8
67	164	1756	0	1920	8
68	163	1756	0	1919	8
69	164	1753	0	1917	8
70	164	1757	0	1921	8
117	164	1759	0	1923	8
118	163	1758	0	1921	8
119	164	1758	0	1922	8
120	164	1759	0	1923	8
121	164	1758	0	1922	8
122	164	1758	0	1922	8
71	164	1756	0	1920	8
73	164	1756	0	1920	8
74	165	1753	0	1918	8
75	165	1752	0	1917	8
78	165	1754	0	1919	8

Timing of Processes for Query - aunt(x y)?

79	165	1762	0	1927	8
134	163	1763	0	1926	8
132	164	1763	0	1927	8
133	163	1763	0	1926	8
135	164	1766	0	1930	8
136	165	8413	3530	12108	30
51	172	1794	0	1966	8
52	164	1773	0	1937	8
54	163	1766	0	1929	8
56	165	8502	3531	12198	30
137	171	1793	0	1964	8
57	163	1757	0	1922	8
63	163	1755	0	1918	8
64	165	1753	0	1918	8
53	164	8494	3581	12239	30
55	171	1798	0	1969	8
76	164	1755	0	1919	8
131	164	1766	0	1930	8
109	163	8498	3603	12264	30
110	171	1792	0	1963	8
113	164	1771	0	1935	8
111	163	1762	0	1925	8
112	164	8495	3605	12264	30
80	172	1791	0	1963	8
81	164	1763	0	1927	8
82	164	8501	3608	12273	30
83	171	1790	0	1961	8
84	164	1768	0	1932	8
85	164	1762	0	1926	8
86	164	1767	0	1931	8
87	163	1771	0	1934	8
90	164	1769	0	1933	8
91	164	8478	3591	12233	30
92	172	1791	0	1963	8
93	164	1761	0	1925	8
38	163	1754	0	1917	8
39	164	1755	0	1919	8
40	164	1752	0	1916	8
108	165	1768	0	1933	8
41	164	1756	0	1920	8
47	165	1753	0	1918	8
89	164	1763	0	1927	8
45	163	1752	0	1915	8
37	164	1754	0	1918	8
43	164	1755	0	1919	8
44	163	1754	0	1917	8
48	164	1753	0	1917	8
160	147	1736	0	1883	7
139	147	1729	0	1876	7
140	148	1730	0	1878	7
141	147	1727	0	1874	7
142	147	1728	0	1875	7
143	148	1728	0	1876	7
144	147	1730	0	1877	7
145	147	1735	0	1882	7
146	147	1734	0	1881	7
147	146	1735	0	1881	7
148	148	1738	0	1886	7
149	147	1935	0	2082	7
150	147	8602	3146	11895	27
151	152	1763	0	1915	7
152	148	1735	0	1883	7
153	147	1736	0	1883	7
154	147	1736	0	1883	7
155	148	1735	0	1883	7
156	147	1737	0	1884	7
157	147	1734	0	1881	7
158	147	1736	0	1883	7
159	148	1735	0	1883	7
182	147	1737	0	1884	7
163	148	1729	0	1877	7
164	147	1729	0	1876	7
165	148	1728	0	1876	7
166	147	1729	0	1876	7
167	147	1735	0	1882	7
168	147	1736	0	1883	7
169	147	1736	0	1883	7
170	147	1736	0	1883	7
171	147	1736	0	1883	7
172	147	1735	0	1882	7
173	147	1937	0	2084	7
174	147	8581	3145	11873	27
175	152	1765	0	1917	7
176	147	1737	0	1884	7
177	147	1737	0	1884	7
178	147	1737	0	1884	7
179	147	1735	0	1882	7
180	147	1737	0	1884	7

Timing of Processes for Query - aunt(x y)? (cont)

181	147	1737	0	1884	7
161	147	1727	0	1874	7
162	147	1727	0	1874	7
204	148	1735	0	1883	7
183	147	1728	0	1875	7
184	147	1729	0	1876	7
187	148	1730	0	1878	7
188	146	1728	0	1874	7
187	147	1735	0	1882	7
190	148	1735	0	1883	7
191	147	1958	0	2105	6
192	147	8608	3147	11902	27
193	152	1762	0	1914	7
194	147	1737	0	1884	7
175	147	1736	0	1883	7
196	147	1734	0	1881	7
197	147	1733	0	1880	7
198	147	1736	0	1883	7
199	147	1735	0	1882	7
200	147	1736	0	1883	7
201	147	1736	0	1883	7
226	147	8670	3184	12001	27
221	152	1771	0	1923	7
248	147	1729	0	1876	7
77	165	1754	0	1919	8
208	147	1727	0	1874	7
233	148	1728	0	1876	7
234	147	1729	0	1876	7
235	147	1728	0	1875	7
236	147	1728	0	1875	7
237	147	1727	0	1874	7
238	148	1728	0	1976	7
239	147	1727	0	1874	7
240	147	1727	0	1874	7
241	147	1728	0	1975	7
242	147	1728	0	1875	7
205	147	1730	0	1877	7
206	147	1728	0	1875	7
207	147	1729	0	1876	7
209	148	1729	0	1877	7
210	147	1727	0	1874	7
245	147	1728	0	1875	7
270	148	1728	0	1876	7
220	148	1742	0	1890	7
243	147	1730	0	1877	7
230	147	8570	3144	11881	27
232	152	1984	0	2136	7
88	163	1763	0	1926	8
227	147	1735	0	1882	7
229	147	1736	0	1883	7
244	147	1727	0	1874	7
211	147	1735	0	1882	7
250	147	8707	3193	12047	27
251	152	1979	0	2131	7
212	147	1735	0	1882	7
213	147	1735	0	1882	7
216	147	1737	0	1884	7
217	148	1735	0	1883	7
218	148	1735	0	1883	7
219	147	1742	·0	1889	7
222	147	1742	0	1889	7
223	147	8683	3198	12028	27
215	152	1765	0	1917	7
247	147	1729	0	1876	7
252	147	1734	0	1881	7
257	147	1729	0	1876	7
259	147	1728	0	1875	7
260	147	1727	0	1874	7
267	147	1728	0	1875	7
269	147	1729	0	1876	7
253	147	1736	0	1883	7
254	147	1735	0	1882	7
255	147	1728	0	1875	7
256	147	1728	0	1875	7
258	147	1728	0	1875	7
261	147	1728	0	1875	7
262	148	1729	0	1877	7
264	147	1728	0	1875	7
265	147	1727	0	1874	7
266	148	1728	0	1876	7
268	147	1729	0	1876	7
263	147	1729	0	1876	7
246	146	1728	0	1874	7
249	147	8581	3146	11974	27
358	152	8772	3231	12155	27
228	153	1764	0	1917	7

Timing of Processes for Query - aunt(x y)? (cont)

355	146	1958	0	2104	6
331	146	8738	3219	12103	27
297	152	1756	0	1908	7
304	147	1737	0	1884	7
330	147	1735	0	1882	7
332	147	1736	0	1883	7
300	147	1736	0	1883	7
277	147	1736	0	1883	7
316	147	1730	0	1877	7
301	147	1737	0	1884	7
302	147	1737	0	1884	7
308	147	1742	0	1889	7
309	147	8681	3180	12008	27
310	152	1974	0	2126	7
311	147	1747	0	1894	7
273	147	1727	0	1874	7
303	147	1735	0	1882	7
272	147	1728	0	1875	7
307	147	1741	0	1888	7
276	147	1729	0	1876	7
326	147	1735	0	1882	7
327	147	1735	0	1882	7
294	147	1730	0	1877	7
296	147	1727	0	1874	7
322	146	1734	0	1880	7
323	147	8684	3181	12012	27
320	152	1757	0	1909	7
321	147	1735	0	1882	7
324	147	1958	0	2105	6
325	147	1739	0	1886	7
328	147	1737	0	1884	7
329	147	1735	0	1882	7
293	147	1728	0	1875	7
295	147	1728	0	1875	7
298	148	1728	0	1876	7
299	147	1737	0	1884	7
333	147	1735	0	1882	7
334	147	1735	0	1882	7
315	147	1735	0	1882	7
319	147	1730	0	1877	7
335	147	1736	0	1883	7
214	147	1735	0	1882	7
275	147	1727	0	1874	7
291	147	1744	0	1891	7
312	147	1743	0	1890	7
279	147	1736	0	1883	7
337	146	1728	0	1874	7
346	147	1736	0	1883	7
202	148	1737	0	1885	7
347	147	1736	0	1883	7
357	147	8627	3145	11919	27
348	152	1764	0	1916	7
351	147	1742	0	1889	7
354	147	1742	0	1889	7
349	147	1735	0	1882	7
356	146	8707	3206	12059	27
350	153	1763	0	1916	7
343	147	1736	0	1883	7
49	164	1754	0	1918	8
340	147	1728	0	1875	7
341	147	1729	0	1876	7
342	147	1729	0	1876	7
344	147	1735	0	1882	7
283	147	1736	0	1883	7
287	147	1742	0	1889	7
288	148	1743	0	1891	7
289	148	1742	0	1890	7
290	147	1741	0	1888	7
345	148	1737	0	1885	7
286	147	8641	3178	11966	27
282	153	1764	0	1917	7
339	148	1730	0	1878	7
281	147	1735	0	1882	7
338	147	1727	0	1874	7
278	147	1735	0	1882	7
280	147	1735	0	1882	7
336	147	1739	0	1886	7
225	147	8693	3203	12043	27
353	152	1767	0	1919	7
285	148	1958	0	2106	7
313	147	1740	0	1887	7
203	146	1736	0	1882	7
292	147	1743	0	1890	7
186	147	1730	0	1877	7
271	147	1729	0	1876	7
359	107	1705	0	1812	5
360	106	1706	0	1812	5
361	106	1705	0	1811	5
362	106	1709	0	1815	5
363	106	1712	0	1818	5

Timing of Processes for Query - aunt(x y)? (cont)

364	107	1712	0	1819	5
365	105	1715	0	1820	5
366	106	1713	0	1819	5
367	106	1713	0	1819	5
368	105	1713	0	1818	5
369	105	1713	0	1818	5
370	106	1712	0	1818	5
371	107	1712	0	1819	5
372	106	1714	0	1820	5
373	106	1713	0	1819	5
374	106	1713	0	1819	5
375	105	1714	0	1819	5
376	106	1712	0	1818	5
377	107	1714	0	1821	5
378	106	1713	0	1819	5
379	107	1713	0	1820	5
331	147	1741	0	1888	7
389	105	1705	0	1810	5
390	106	1711	0	1817	5
391	106	1714	0	1820	5
392	106	1715	0	1821	5
393	106	1714	0	1820	5
394	106	1714	0	1820	5
395	106	1713	0	1819	5
396	106	1712	0	1818	5
397	106	1712	0	1818	5
398	106	1714	0	1820	5
399	106	1713	0	1819	5
400	106	1712	0	1818	5
401	106	1713	0	1819	5
381	107	1714	0	1921	5
382	106	1713	0	1819	5
383	106	1713	0	1819	5
384	106	1713	0	1819	5
385	106	1712	0	1818	5
386	105	1713	0	1818	5
387	107	1713	0	1820	5
388	106	1712	0	1818	5
380	106	1712	0	1818	5
305	147	1742	0	1889	7
411	106	1706	0	1812	5
412	105	1708	0	1813	5
413	105	1714	0	1819	5
414	106	1712	0	1818	5
415	105	1714	0	1819	5
416	106	1713	0	1819	5
417	106	1714	0	1820	5
418	105	1715	0	1820	5
419	107	1714	0	1821	5
420	106	1714	0	1820	5
421	105	1714	0	1819	5
422	106	1712	0	1818	5
423	106	1713	0	1819	5
402	106	1713	0	1819	5
446	106	1718	0	1824	5
433	106	1715	0	1821	5
434	106	1712	0	1818	5
435	106	1713	0	1819	5
436	106	1714	0	1820	5
437	106	1715	0	1821	5
438	106	1712	0	1818	5
439	105	1714	0	1819	5
440	105	1713	0	1818	5
441	106	1713	0	1819	5
442	106	1712	0	1818	5
443	106	1713	0	1819	5
444	106	1713	0	1819	5
445	106	1714	0	1820	5
407	106	1713	0	1819	5
432	106	1714	0	1820	5
403	106	1714	0	1820	5
404	106	1714	0	1820	5
406	105	1714	0	1819	5
408	106	1712	0	1818	5
409	106	1714	0	1820	5
410	106	1713	0	1819	5
405	107	1712	0	1819	5
429	106	1712	0	1818	5
430	106	1714	0	1820	5
431	106	1713	0	1819	5
317	147	1743	0	1890	7
425	106	1705	0	1811	5
426	107	1710	0	1817	5
427	106	1714	0	1820	5
428	105	1713	0	1818	5
448	106	1712	0	1818	5
449	106	1714	0	1820	5
450	106	1714	0	1820	5

Timing of Processes for Query - aunt(x y)? (cont)

451	107	1713	0	1820	5
452	106	1713	0	1819	5
453	106	1720	0	1826	5
424	105	1713	0	1818	5
42	164	1763	0	1927	8
447	106	1708	0	1814	5
534	106	1722	0	1828	5
521	107	1722	0	1829	5
522	106	1721	0	1827	5
523	107	1533	0	1640	6
524	106	1535	0	1641	6
500	105	1721	0	1826	5
501	105	1718	0	1823	5
502	106	1719	0	1825	5
503	106	1717	0	1823	5
504	106	1716	0	1822	5
505	107	1718	0	1825	5
506	106	1718	0	1824	5
507	106	1716	0	1822	5
508	105	1717	0	1822	5
274	147	1735	0	1882	7
490	107	1720	0	1827	5
318	148	1741	0	1889	7
306	147	1736	0	1883	7
550	106	1709	0	1815	5
551	107	1721	0	1828	5
552	106	1721	0	1827	5
558	105	1719	0	1823	5
539	106	1719	0	1825	5
540	106	1717	0	1823	5
541	105	1719	0	1824	5
553	107	1721	0	1828	5
542	106	1721	0	1827	5
471	105	1716	0	1821	5
536	106	1720	0	1826	5
537	107	1722	0	1829	5
561	106	1715	0	1821	5
562	105	1716	0	1821	5
472	106	1717	0	1823	5
474	106	1717	0	1823	5
475	106	1725	0	1831	5
473	105	1717	0	1822	5
574	107	1530	0	1637	6
575	105	1736	0	1841	5
576	106	1730	0	1836	5
577	106	1732	0	1838	5
469	106	1718	0	1824	5
470	106	1717	0	1823	5
476	106	1724	0	1830	5
534	106	1719	0	1825	5
573	106	1528	0	1634	6
547	106	1723	0	1829	5
560	106	1715	0	1821	5
559	106	1714	0	1820	5
564	106	1726	0	1832	5
565	106	1722	0	1828	5
566	106	1723	0	1829	5
567	106	1722	0	1828	5
568	106	1724	0	1830	5
569	105	1725	0	1830	5
570	105	1725	0	1830	5
571	105	1729	0	1834	5
572	106	1730	0	1836	5
543	106	1721	0	1827	5
548	106	1719	0	1825	5
538	106	1721	0	1827	5
546	107	1721	0	1828	5
622	106	1717	0	1823	5
557	107	1719	0	1826	5
535	106	1720	0	1826	5
588	107	1717	0	1824	5
610	107	1721	0	1828	5
603	106	1721	0	1827	5
604	105	1721	0	1826	5
615	105	1718	0	1823	5
602	106	1717	0	1823	5
611	106	1720	0	1826	5
612	107	1721	0	1828	5
613	105	1718	0	1823	5
587	106	1716	0	1822	5
605	106	1717	0	1823	5
594	105	1718	0	1823	5
618	106	1717	0	1823	5

Timing of Processes for Query - aunt(x y)? (cont)

620	106	1715	0	1821	5
595	106	1718	0	1824	5
596	106	1718	0	1824	5
597	106	1718	0	1824	5
477	106	1725	0	1831	5
578	106	1716	0	1822	5
478	106	1724	0	1830	5
592	106	1714	0	1820	5
593	107	1716	0	1823	5
590	106	1715	0	1821	5
591	106	1714	0	1820	5
589	106	1717	0	1823	5
599	107	1716	0	1823	5
614	107	1721	0	1828	5
608	106	1719	0	1825	5
617	106	1714	0	1820	5
616	106	1717	0	1823	5
545	105	1720	0	1825	5
579	106	1715	0	1821	5
581	106	1715	0	1821	5
580	105	1718	0	1823	5
582	106	1715	0	1821	5
583	106	1718	0	1824	5
584	107	1716	0	1823	5
585	106	1718	0	1824	5
606	106	1718	0	1824	5
609	106	1720	0	1826	5
621	106	1715	0	1821	5
586	106	1715	0	1921	5
601	106	1720	0	1826	5
607	106	1719	0	1825	5
619	106	1716	0	1822	5
563	106	1724	0	1830	5
544	106	1720	0	1826	5
623	107	1720	0	1827	5
657	106	1720	0	1826	5
549	106	1721	0	1827	5
645	105	1718	0	1823	5
672	105	1718	0	1823	5
690	105	1722	0	1827	5
706	106	1720	0	1826	5
661	107	1720	0	1827	5
662	106	1721	0	1827	5
663	106	1720	0	1826	5
673	107	1719	0	1826	5
674	106	1721	0	1827	5
675	107	1719	0	1826	5
664	106	1720	0	1826	5
675	105	1718	0	1823	5
651	106	1721	0	1827	5
693	106	1721	0	1827	5
652	106	1720	0	1826	5
653	106	1720	0	1926	5
654	107	1720	0	1827	5
649	106	1719	0	1825	5
655	106	1720	0	1826	5
658	106	1720	0	1826	5
659	106	1720	0	1826	5
650	106	1720	0	1826	5
647	106	1721	0	1827	5
648	106	1720	0	1826	5
646	106	1721	0	1827	5
656	106	1718	0	1824	5
676	105	1719	0	1824	5
692	107	1719	0	1826	5
691	105	1720	0	1825	5
670	105	1719	0	1824	5
668	106	1721	0	1827	5
698	105	1718	0	1823	5
700	106	1721	0	1827	5
701	105	1720	0	1825	5
697	106	1717	0	1823	5
699	107	1720	0	1827	5
702	107	1720	0	1827	5
703	106	1721	0	1827	5
704	106	1720	0	1826	5
705	106	1721	0	1827	5
667	106	1721	0	1827	5
671	105	1718	0	1823	5
660	106	1720	0	1826	5
669	106	1718	0	1824	5
694	105	1719	0	1824	5
696	105	1718	0	1823	5
665	105	1718	0	1823	5
689	107	1722	0	1929	5
707	106	1720	0	1826	5

Timing of Processes for Query - aunt(x y)? (cont)

708	105	1720	0	1825	5
630	105	1717	0	1822	5
491	106	1715	0	1821	5
624	106	1720	0	1826	5
629	106	1717	0	1823	5
479	106	1725	0	1831	5
480	105	1725	0	1830	5
481	107	1724	0	1831	5
492	106	1718	0	1824	5
494	105	1716	0	1821	5
493	106	1718	0	1824	5
482	105	1724	0	1829	5
628	106	1715	0	1821	5
636	106	1716	0	1822	5
627	105	1717	0	1822	5
637	105	1718	0	1823	5
638	106	1717	0	1823	5
639	106	1716	0	1822	5
634	105	1715	0	1820	5
640	106	1717	0	1823	5
642	106	1715	0	1821	5
643	106	1716	0	1822	5
635	106	1718	0	1824	5
632	106	1716	0	1822	5
633	105	1716	0	1821	5
631	107	1717	0	1824	5
641	106	1718	0	1824	5
468	106	1724	0	1830	5
625	106	1716	0	1822	5
626	106	1715	0	1821	5
487	107	1732	0	1839	5
680	106	1717	0	1823	5
485	107	1529	0	1636	6
682	106	1721	0	1827	5
683	106	1717	0	1823	5
679	106	1717	0	1823	5
681	106	1717	0	1823	5
684	106	1715	0	1821	5
685	106	1715	0	1821	5
686	106	1718	0	1824	5
687	106	1717	0	1823	5
483	106	1731	0	1837	5
489	106	1731	0	1837	5
690	107	1716	0	1823	5
486	106	1528	0	1634	6
677	106	1721	0	1827	5
678	106	1717	0	1823	5
484	107	1731	0	1838	5
705	106	1719	0	1825	5
533	105	1717	0	1822	5
[illegible]	107	1731	0	1838	5
466	107	1719	0	1826	5
461	107	1722	0	1829	5
495	106	1716	0	1822	5
709	106	1712	0	1818	5
464	107	1722	0	1829	5
462	106	1723	0	1829	5
[illegible]	106	1533	0	1639	6
459	106	1537	0	1643	6
477	105	1724	0	1829	5
463	106	1721	0	1827	5
460	106	1722	0	1828	5
499	106	1713	0	1819	5
515	106	1714	0	1820	5
498	106	1713	0	1819	5
516	106	1714	0	1820	5
518	106	1713	0	1819	5
513	106	1714	0	1820	5
517	106	1713	0	1819	5
454	106	1722	0	1828	5
644	107	1714	0	1821	5
455	106	1722	0	1828	5
456	107	1722	0	1829	5
514	105	1714	0	1819	5
510	106	1714	0	1820	5
511	107	1713	0	1820	5
467	107	1722	0	1829	5
512	107	1713	0	1820	5
519	105	1719	0	1824	5
185	147	1735	0	1882	7
497	105	1707	0	1812	5
496	106	1710	0	1816	5
526	106	1722	0	1828	5
529	106	1720	0	1826	5
46	164	1765	0	1929	8
525	106	1715	0	1821	5
528	106	1718	0	1824	5

Timing of Processes for Query - aunt(x y)? (cont)

531	105	1720	0	1825	5
527	106	1722	0	1828	5
530	106	1719	0	1825	5
532	106	1723	0	1829	5
533	106	1722	0	1828	5
224	147	1966	0	2113	6
520	106	1713	0	1819	5
352	147	1750	0	1897	7
284	147	1738	0	1885	7
688	107	1706	0	1813	5
666	106	1708	0	1814	5
578	106	1730	0	1836	5
556	106	1713	0	1819	5
710	106	1714	0	1820	5
465	106	1719	0	1825	5
314	146	1747	0	1893	7

Times in microsecs

Total No Procs = 711; Spawning Procs = 33; Non Spawning Procs = 678
Average Set Up Time (all procs) = 129; Average Rewrite Time (all procs) = 2044
Average Rewrite Time (non spawning) = 1732
Average Set Up Time (non spawning) = 129
Average Rewrite Time (spawning) = 8465
Average Set Up Time (spawning)= 140
Average Spawn Time = 3327
Percentage of Spawning/Non Spawning Procs = 4

Timing of Processes for Query - aunt(x y)? (cont)

G5. Function Call Details

The following tables show details from several processes for the queries
"aunt(x y)?"
and
"stepparent(x y)?".

Function Categories

1	Top level "eval" functions
2	Lower level list/exp "eval" functions
3	Lower level rule "eval" functions
4	Lower level arithmetical functions
5	Variable installation and instantiation functions
6	Process related functions incl. spawning functions
7	Memory space creation functions
8	Garbage collection functions

Function Category Reference List

Process	Function Category								
	1	2	3	4	5	6	7	8	Total
No.6 Spawning									
Set-up	–	–	–	–	–	2	2		4
Rewrite	50	4	184	–	1	–	259	12	510
Spawn	–	–	–	–	–	120	12	–	132
Total									646
No. 116 Non Spawn									
Set-up	–	–	–	–	–	5	3	–	8
Rewrite	23	6	–	–	5	–	20	46	100
Spawn	–	–	–	–	–	–	–	–	–
Total									108
No. 284 Spawning									
Set-up	–	–	–	–	–	4	2	–	6
Rewrite	181	6	163	–	2	–	243	29	624
Spawn	–	–	–	–	–	112	47	–	159
Total									789
No. 285 Non Spawn									
Set-up	–	–	–	–	–	4	2	–	6
Rewrite	23	6	–	–	6	–	19	43	97
Spawn	–	–	–	–	–	–	–	–	–
Total									103

Query - aunt(x y)?

Function Call Details for query - aunt(x y)?

Process	Function Category								
	1	2	3	4	5	6	7	8	Total
No.0 Spawning									
Set-up	–	–	–	–	–	2	2		4
Rewrite	9	–	37	–	–	–	41	–	87
Spawn	--	–	–	–	–	18	6	–	24
Total									115
No. 12 Spawning									
Set-up	–	–	–	–	–	3	2	–	5
Rewrite	54	3	163	–	2	–	239	21	482
Spawn	–	–	–	–	–	114	27	–	141
Total									628
No. 79 NonSpawn									
Set-up	–	–	–	–	–	3	1	–	4
Rewrite	23	6	–	–	6	–	23	43	101
Spawn	–	–	–	–	–	–	–	–	--
Total									105
No. 210 Non Spawn									
Set-up	–	–	–	–	–	3	1	–	4
Rewrite	506	138	164	–	6	–	710	602	2126
Spawn	–	–	–	–	–	–	–	–	--
Total									2130

Query - stepparent(x y)?

Function Call Details for query - stepparent(x y)?

G6. Bus Usage Results

These results refer to the use of the broadcast busses during the evaluation of the query "firstcousin(x y)?" The figures show the amount of contention for the broadcast system and the resulting delays.

BUS TIMES

Bus No.	Sender PE	Start Time	Packet Size	No.Procs	Delay
0	0	12972	15	21	0
1	0	25290	14	21	0
2	16	25303	14	21	0
3	21	25303	14	21	0
4	1	25304	14	21	0
0	5	25304	14	21	0
1	7	25305	14	21	1
2	8	25318	14	21	14
3	11	25318	14	21	14
4	12	25319	14	21	15
0	13	25319	14	21	15
1	18	25320	14	21	16
2	17	25333	14	21	28
3	6	25333	14	21	27
4	9	25334	14	21	28
0	20	25334	14	21	28
1	2	25335	14	21	28
2	3	25348	14	21	41
3	4	25348	14	21	41
4	14	25349	14	21	42
0	15	25349	14	21	42
1	10	25350	14	21	42
2	19	25363	14	21	55
1	16	38108	15	21	0
2	19	38301	15	21	0
3	15	40306	15	21	0
4	17	40414	15	21	0
0	22	46283	15	21	0
1	21	46319	15	21	0
2	44	48279	15	21	0
3	43	48335	15	21	0
4	9	48371	15	21	0
0	8	48374	15	21	0
1	16	51126	15	21	0
2	4	51531	13	21	0
3	5	51555	13	21	0
4	37	51555	13	21	0
0	3	51570	13	21	0
1	36	51595	13	21	0
2	35	51640	13	21	0
3	14	53551	13	21	0
4	13	53620	13	21	0
0	46	53663	13	21	0
1	18	53693	13	21	0
2	45	53733	13	21	0
3	47	53733	13	21	0
4	30	59453	13	21	0
0	24	59536	13	21	0
1	32	59662	13	21	0
2	25	59683	13	21	0
3	23	59698	13	21	0
4	31	59782	13	21	0
0	10	61586	13	21	0
1	33	61744	13	21	0
2	11	61749	13	21	0
3	34	61818	13	21	0
4	9	62755	13	21	0
0	1	63757	13	21	0
1	0	63761	13	21	0
2	49	63817	13	21	0
3	32	91915	13	21	0
4	0	95947	13	21	0
0	38	99474	15	21	0
1	0	112385	13	21	0
2	37	168927	15	21	0
3	43	171944	15	21	0
4	41	181231	13	21	0
0	40	181288	13	21	0
1	39	181289	13	21	0
2	37	185171	13	21	0
3	34	185317	13	21	0
4	38	185408	13	21	0
0	48	213481	13	21	0
1	2	223929	13	21	0
2	44	238755	15	21	0
3	1	251004	13	21	0
4	4	251451	13	21	0
0	44	253145	13	21	0
1	49	350999	13	21	0
2	6	358070	13	21	0
3	7	358818	13	21	0
4	5	380909	13	21	0
0	18	385459	13	21	0
1	45	613605	15	21	0
2	17	625896	13	21	0
3	15	625978	13	21	0
4	16	625979	13	21	0

Times in microsecs

Data on Bus Usage

50 PE/2 Bus Configuration (original version)

BUS TIMES

Bus No.	Sender PE	Start Time	Packet Size	No.Procs	Delay
0	0	12973	15	21	0
1	0	25291	14	21	0
0	5	25303	14	21	0
1	8	25306	14	21	3
0	21	25318	14	21	15
1	9	25321	14	21	17
0	12	25333	14	21	28
1	14	25336	14	21	31
0	16	25348	14	21	43
1	20	25351	14	21	46
0	13	25363	14	21	57
1	17	25366	14	21	60
0	18	25378	14	21	72
1	19	25381	14	21	75
0	1	25393	14	21	86
1	4	25396	14	21	89
0	10	25408	14	21	101
1	2	25411	14	21	103
0	6	25423	14	21	115
1	7	25426	14	21	118
0	15	25438	14	21	130
1	3	25441	14	21	132
0	11	25453	14	21	144
1	16	38127	15	21	0
0	19	38281	15	21	0
1	38	48265	15	21	0
0	28	48284	15	21	0
1	27	48286	15	21	0
0	37	48305	15	21	0
1	26	48363	15	21	0
0	15	48366	15	21	0
1	14	48390	15	21	0
0	46	50270	15	21	0
1	43	50346	15	21	0
0	7	51474	13	21	0
1	5	51516	13	21	0
0	6	51572	13	21	0
1	36	51585	13	21	0
0	39	51610	13	21	0
1	40	51618	13	21	0
0	22	54363	15	21	0
1	21	54529	15	21	0
0	23	54531	15	21	0
1	17	58086	15	21	0
0	31	60475	13	21	0
1	29	60634	13	21	0
0	30	60637	13	21	0
1	10	62461	13	21	0
0	37	62594	13	21	0
1	9	62666	13	21	0
0	34	62683	13	21	0
1	8	62747	13	21	0
0	35	62780	13	21	0
1	46	66804	13	21	0
0	18	66869	13	21	0
1	44	66918	13	21	0
0	38	66937	13	21	0
1	15	66993	13	21	0
0	45	67016	13	21	0
1	39	68420	13	21	0
0	16	68991	13	21	0
1	33	70925	13	21	0
0	32	71044	13	21	0
1	40	72361	13	21	0
0	37	74829	13	21	0
1	49	99956	13	21	0
0	42	103803	13	21	0
1	43	103941	13	21	0
0	41	104095	13	21	0
1	18	107330	13	21	0
0	9	149890	13	21	0
1	9	166273	13	21	0
0	10	235709	13	21	0
1	30	237625	13	21	0
0	10	252259	13	21	0
1	15	276225	15	21	0
0	0	281008	13	21	0
1	20	287494	13	21	0
0	7	288534	13	21	0
1	3	288589	13	21	0
0	6	288589	13	21	0
1	31	363279	13	21	0
0	48	365475	13	21	0
1	32	370949	13	21	0
0	13	427499	13	21	0
1	8	494010	13	21	0
0	8	644351	13	21	0

Times in microsecs

Data on Bus Usage

20 PE/5 Bus Configuration (original version)

BUS TIMES

Bus No.	Sender PE	Start Time	Packet Size	No.Procs	Delay
0	0	12967	15	21	0
1	0	25289	14	21	0
2	2	25297	14	21	0
3	11	25298	14	21	0
4	18	25298	14	21	0
0	4	25299	14	21	0
1	6	25304	14	21	5
2	9	25312	14	21	13
3	14	25313	14	21	14
4	16	25313	14	21	14
0	7	25314	14	21	14
1	10	25319	14	21	19
2	15	25327	14	21	27
3	12	25328	14	21	27
4	13	25328	14	21	27
0	19	25329	14	21	28
1	1	25334	14	21	33
2	5	25342	14	21	40
3	3	25343	14	21	40
4	8	25343	14	21	40
0	17	25344	14	21	41
1	16	38350	15	21	0
2	6	40319	15	21	0
3	5	40364	15	21	0
4	2	50332	15	21	0
0	1	50373	15	21	0
1	7	51613	13	21	0
2	3	51677	13	21	0
3	4	51764	13	21	0
4	19	53723	13	21	0
0	0	53780	13	21	0
1	17	55799	13	21	0
2	15	55860	13	21	0
3	14	55866	13	21	0
4	18	60568	15	21	0
0	1	62674	13	21	0
1	16	62810	13	21	0
2	13	74743	13	21	0
3	1	133281	14	21	0
4	3	172799	15	21	0
0	12	183269	15	21	0
1	1	185109	13	21	0
2	0	185161	13	21	0
3	3	187140	13	21	0
4	1	197380	13	21	0
0	15	208681	13	21	0
1	14	211027	13	21	0
2	17	230863	13	21	0
3	5	234528	13	21	0
4	17	250461	15	21	0
0	11	261074	15	21	0
1	13	261491	15	21	0
2	3	263768	13	21	0
3	4	264672	13	21	0
4	5	275228	13	21	0
0	4	285881	13	21	0
1	5	309857	13	21	0
2	15	366085	13	21	0
3	19	376593	15	21	0
4	6	384554	13	21	0
0	14	389621	13	21	0
1	13	444930	13	21	0
2	5	497257	15	21	0
3	18	501213	13	21	0
4	7	504272	13	21	0
0	13	511613	13	21	0
1	9	524681	13	21	0
2	3	568115	13	21	0
3	4	582370	13	21	0
4	4	614027	15	21	0
0	12	622701	13	21	0
1	14	624158	13	21	0
2	11	626348	13	21	0
3	16	627370	13	21	0
4	3	633009	13	21	0
0	4	635312	15	21	0
1	4	649760	13	21	0
2	5	660300	13	21	0
3	2	697881	14	21	0
4	6	699576	13	21	0
0	19	705474	15	21	0
1	15	718049	13	21	0
2	18	718807	13	21	0
3	9	719144	13	21	0
4	0	722439	13	21	0
0	13	877302	13	21	0
1	10	885227	13	21	0

Times in microsecs

Data on Bus Usage

20 PE/2 Bus Configuration (original version)

BUS TIMES

Bus No.	Sender PE	Start Time	Packet Size	No.Procs	Delay
0	0	12967	15	21	0
1	0	25290	14	21	0
0	2	25296	14	21	0
1	9	25305	14	21	6
0	15	25311	14	21	12
1	18	25320	14	21	21
0	19	25326	14	21	27
1	4	25335	14	21	35
0	10	25341	14	21	40
1	13	25350	14	21	49
0	1	25356	14	21	55
1	3	25365	14	21	63
0	6	25371	14	21	69
1	8	25380	14	21	78
0	11	25386	14	21	84
1	16	25395	14	21	93
0	17	25401	14	21	99
1	14	25410	14	21	107
0	7	25416	14	21	112
1	12	25425	14	21	121
0	5	25431	14	21	126
1	19	38414	15	21	0
0	2	50299	15	21	0
1	1	50345	15	21	0
0	5	51605	13	21	0
1	4	51675	13	21	0
0	6	51677	13	21	0
1	17	54457	15	21	0
0	18	54475	15	21	0
1	19	63478	15	21	0
0	9	63699	13	21	0
1	8	63760	13	21	0
0	7	63768	13	21	0
1	3	68015	13	21	0
0	5	68460	13	21	0
1	4	68759	13	21	0
0	5	91901	13	21	0
1	17	131300	15	21	0
0	5	145588	13	21	0
1	13	147247	13	21	0
0	17	147753	13	21	0
1	10	152727	13	21	0
0	15	217583	13	21	0
1	14	219413	13	21	0
0	12	223624	13	21	0
1	11	223768	13	21	0
0	4	230704	13	21	0
1	16	250836	13	21	0
0	15	263445	15	21	0
1	9	276029	13	21	0
0	8	277920	13	21	0
1	15	278008	13	21	0
0	9	323041	15	21	0
1	9	337371	13	21	0
0	3	431810	13	21	0
1	5	456681	13	21	0
0	10	472943	13	21	0
1	10	500340	15	21	0
0	0	512624	13	21	0
1	17	512701	13	21	0
0	7	536260	15	21	0
1	7	550510	13	21	0
0	16	562145	15	21	0
1	12	574409	13	21	0
0	11	574508	13	21	0
1	16	585222	15	21	0
0	8	593080	15	21	0
1	7	597705	13	21	0
0	8	605329	13	21	0
1	8	630752	15	21	0
0	13	642999	13	21	0
1	12	643115	13	21	0
0	2	644210	13	21	0
1	1	673455	14	21	0
0	2	675676	14	21	0
1	10	676581	13	21	0
0	3	678858	13	21	0
1	16	707410	15	21	0
0	18	707580	13	21	0
1	14	716426	13	21	0
0	7	727385	13	21	0
1	19	728274	13	21	0
0	0	729624	13	21	0
1	0	750688	13	21	0
0	19	753480	13	21	0
1	16	864908	13	21	0
0	14	1012149	13	21	0

Times in microsecs

Data on Bus Usage

G7. Input Memory Utilisation

These tables refer to the maximum number of words held in each input memory during the course of the evaluation of the query "firstcousin(x y)?".

QUERY -firstcousin(x y)?

50 PE/5 Bus Configuration (original version)

INPUT MEMORY DATA

Bus No.0 Max.No.Words	Bus No.1 Max.No.Words	Bus No.2 Max.No.Words	Bus No.3 Max.No.Words	Bus No.4 Max.No.Words
30	56	55	41	42
41	56	55	41	42
30	56	54	26	56
41	56	40	30	56
28	56	53	13	56
26	42	30	26	56
67	42	41	26	56
67	42	55	26	56
52	42	40	26	56
39	42	29	13	56
67	26	42	30	56
54	26	56	15	56
41	26	56	26	56
41	26	56	26	42
54	26	42	15	42
26	41	57	26	42
55	26	57	13	42
39	41	57	26	42
39	40	56	39	40
68	28	56	26	39
42	26	56	54	26
43	41	42	40	40
43	41	56	40	26
56	26	56	39	41
56	39	56	39	28
56	26	56	28	41
56	26	55	54	15
56	26	55	54	15
56	26	41	68	26
56	15	54	68	15
56	52	28	55	28
56	26	26	56	54
56	39	15	56	41
56	39	41	56	26
42	26	41	68	26
42	26	39	56	26
42	54	26	56	13
42	41	26	56	39
42	15	41	56	26
30	28	41	56	26
26	42	54	56	26
30	42	54	56	26
30	42	54	55	26
26	28	40	42	53
26	42	40	42	53
13	42	40	42	39
26	42	40	42	39
13	42	53	42	39
30	42	41	54	28
30	42	54	41	42

Input Memory Values for Each Processing Element

QUERY - firstcousin(x y)?

50 PE/2 Bus Configuration (original version)

INPUT MEMORY DATA

Bus No.0 Max.No.Words	Bus No.1 Max.No.Words
95	71
95	69
82	82
82	82
82	82
80	66
79	56
79	56
82	70
82	70
95	84
82	84
82	84
82	71
56	82
70	93
84	80
82	69
69	82
82	70
69	84
56	95
70	68
84	81
84	110
84	123
71	92
70	107
70	94
84	95
70	108
84	95
84	73
84	69
70	67
99	56
96	56
93	56
95	56
94	55
79	68
84	67
84	67
84	67
97	56
106	56
121	56
97	43
97	42
82	69

Input Memory Values for Each Processing Element

QUERY -firstcousin(x y)?

20 PE/2 Bus Configuration (original version)

INPUT MEMORY DATA

Bus No.0 Max.No.Words	Bus No.1 Max.No.Words
126	152
155	140
169	138
154	138
140	167
154	129
154	129
154	151
154	138
140	126
154	126
154	126
154	126
154	126
154	126
150	126
154	126
154	127
140	138
151	165

QUERY - firstcousin(x y)?

20 PE/5 Bus Configuration (original version)

INPUT MEMORY DATA

Bus No.0 Max.No.Words	Bus No.1 Max.No.Words	Bus No.2 Max.No.Words	Bus No.3 Max.No.Words	Bus No.4 Max.No.Words
81	57	82	81	72
97	56	67	96	70
85	82	81	72	70
81	69	81	83	82
56	109	56	83	68
82	67	81	56	85
72	81	81	56	70
56	96	81	57	85
72	58	68	67	86
58	58	81	58	58
69	58	81	57	71
69	54	81	57	71
83	57	81	57	71
72	54	81	57	71
56	54	96	81	57
56	43	96	68	70
73	81	42	68	68
97	54	95	68	70
98	80	58	68	70
82	43	82	68	85

Input Memory Values for Each Processing Element

QUERY - firstcousin(x y)?

50 PE/5 Bus Configuration (optimised version)

INPUT MEMORY DATA

Bus No.0 Max.No.Words	Bus No.1 Max.No.Words	Bus No.2 Max.No.Words	Bus No.3 Max.No.Words	Bus No.4 Max.No.Words
27	42	40	39	54
15	42	42	54	56
15	42	28	26	56
26	42	28	26	56
15	28	54	67	56
26	39	43	26	56
26	39	43	26	56
39	28	42	26	56
52	28	42	54	56
15	39	56	69	56
15	26	56	69	56
26	39	56	39	42
29	39	56	54	28
29	39	56	41	28
29	39	56	55	28
28	26	56	53	28
57	26	56	41	28
56	26	56	54	26
56	26	42	54	39
56	39	42	54	39
56	26	42	54	39
66	28	42	28	28
42	68	42	56	26
42	68	42	56	28
42	55	42	56	26
43	53	28	42	26
55	15	28	83	26
42	40	28	42	26
42	53	41	42	26
42	40	54	54	13
42	40	54	56	15
42	26	54	56	26
42	28	54	56	26
28	28	67	56	26
28	28	54	56	26
28	29	54	56	26
28	40	54	56	26
28	53	54	56	26
14	42	54	56	41
15	42	54	56	39
15	42	54	42	39
15	42	54	28	39
26	42	54	28	39
26	42	55	28	67
26	42	55	28	67
26	42	55	28	67
26	43	40	28	42
41	43	29	26	42
13	42	29	67	42
26	42	40	39	42

Input Memory Values for Each Processing Element

QUERY - firstcousin(x y)?

50 PE/2 Bus Configuration (optimised version)

INPUT MEMORY DATA

Bus No.0 Max.No.Words	Bus No.1 Max.No.Words
123	56
119	56
121	70
121	70
108	70
80	93
82	80
81	81
68	79
93	91
105	84
109	84
94	84
94	93
78	106
93	106
97	67
96	81
109	70
109	91
83	91
81	78
67	92
70	107
70	120
95	79
67	79
78	78
70	78
70	80
70	80
84	104
84	105
84	94
70	105
70	94
65	109
65	93
70	110
84	123
64	123
84	67
80	67
78	54
78	56
91	84
104	64
70	97
70	65
95	56

Input Memory Values for Each Processing Element

QUERY - firstcousin(x y)?

20 PE/5 Bus Configuration (optimised version)

INPUT MEMORY DATA

Bus No.0 Max.No.Words	Bus No.1 Max.No.Words	Bus No.2 Max.No.Words	Bus No.3 Max.No.Words	Bus No.4 Max.No.Words
84	68	109	96	109
97	54	95	123	94
99	81	86	110	81
68	95	95	110	96
95	68	98	83	84
94	95	110	95	83
81	95	82	110	81
56	96	98	83	110
80	83	85	96	84
56	109	111	83	84
69	109	84	83	84
82	109	97	83	69
82	95	83	87	56
82	96	96	87	56
69	109	109	83	82
69	96	94	83	97
98	109	108	81	96
83	55	94	96	111
122	56	81	124	96
84	54	109	96	109

QUERY - firstcousin(x y)?

20 PE/2 Bus Configuration (optimised version)

INPUT MEMORY DATA

Bus No.0 Max.No.Words	Bus No.1 Max.No.Words
247	165
249	164
238	150
180	166
180	181
179	151
166	151
139	151
139	236
152	223
164	236
179	177
168	218
194	192
181	205
209	190
182	151
194	167
181	182
247	151

Input Memory Values for Each Processing Element

G7. Return of Results

Details are given of the pattern of results return for the query "firstcousin(x y)?" under different machine configurations.

QUERY - firstcousin(x y)?

50 PE/5 Bus Configuration (original version)

TIMES FOR RETURN OF RESULTS

Proc.No.	Time result found	Size of Packet	Pe.No.
: 788	189739	4	29
: 935	191760	4	2
: 919	192155	4	36
: 920	192381	4	35
:1001	194013	4	12
: 898	195218	4	22
:1098	197355	4	15
:1097	197711	4	16
:1177	198467	4	27
:1178	198638	4	26
:1221	198672	4	46
:1315	200424	4	23
:1265	201140	4	9
:1251	203040	4	14
:1266	204184	4	8
:1253	205316	4	7
:1002	206889	4	11
:1112	207499	4	20
:1111	208426	4	21
: 936	208905	4	1
: 787	211028	4	30
:1031	213079	4	17
: 847	213429	4	24
: 760	215413	4	44
:1342	228133	4	32
:1339	230167	4	19
:1340	230338	4	18
:1341	230369	4	4
:1408	248387	4	0
:1067	251393	4	47
:1406	252892	4	28
:1405	252979	4	31
:1407	252932	4	25
:1518	317310	4	39
:1516	319560	4	33
:1222	323292	4	45
:1535	326226	4	42
:1536	326324	4	41
:1517	328456	4	40
:1515	330575	4	34
:1316	331498	4	22
:1491	332330	4	37
:1252	337336	4	9
:1099	341188	4	23
:1068	341576	4	46
:1033	344705	4	15
:1032	345027	4	16
: 832	345710	4	12
: 897	345932	4	26
:1606	349618	4	48
: 831	350248	4	20
:1604	356031	4	43
:1605	356279	4	38
:1603	356348	4	29
:1034	357881	4	30
:1628	362259	4	2
:1625	364421	4	14
:1626	364459	4	11
:1627	364478	4	5
:1573	366345	4	19
:1574	366573	4	18
:1690	389158	4	44
:1689	391479	4	1
:1667	395840	4	27
:1668	396065	4	24
:1492	466554	4	34
: 759	468119	4	45
: 848	481750	4	23
:1254	482505	4	9
:1733	486808	4	10
:1734	487208	4	8
:1755	493752	4	13
:1756	494211	4	12
:1793	516953	4	17
:1794	517119	4	16
:1100	620088	4	23
:1859	761779	4	10
:1860	762148	4	9
:1888	762312	4	49
:1887	764217	4	0

Total query evaluation time = 764352 microsecs

Data on Results Return

QUERY - firstcousin(x y)?

50 PE/2 Bus Configuration (original version)

TIMES FOR RETURN OF RESULTS

Proc.No.	Time result found	Size of Packet	Pe.No.
: 825	188394	4	13
: 826	198578	4	12
: 810	189960	4	36
: 787	193745	4	47
:1097	[illegible]	4	0
:1162	199435	4	21
:1293	202218	4	35
:1294	202267	4	34
:1310	205790	4	24
:1309	205843	4	25
:1333	205952	4	48
:1383	207367	4	46
:1391	207579	4	45
:1360	208125	4	20
:1359	208399	4	23
:1395	209402	4	32
:1338	209730	4	16
:1200	214887	4	3
:1199	214915	4	4
:1178	216059	4	5
:1177	216055	4	6
:1098	216691	4	30
:1120	216617	4	29
:1046	218172	4	7
:1045	219072	4	10
:1031	221380	4	27
:1032	221376	4	26
:1516	242738	4	11
:1517	246878	4	41
:1535	247179	4	2
:1536	247290	4	1
:1515	248044	4	18
:1485	249355	4	19
:1161	272993	4	22
:1557	290381	4	43
:1558	290452	4	42
:1579	306672	4	49
:1580	306935	4	40
:1228	331156	4	47
: 760	337939	4	12
:1099	341196	4	8
:1119	342285	4	31
:1227	344338	4	48
:1121	348138	4	34
: 809	348165	4	37
:1033	350926	4	25
: 979	354439	4	5
:1386	354405	4	32
: 788	354691	4	46
: 980	355522	4	4
:1034	365458	4	29
:1486	384246	4	18
:1518	385360	4	41
:1639	388345	4	35
:1640	388468	4	33
: 759	409313	4	13
:1683	416939	4	28
:1684	417272	4	27
:1705	423390	4	36
:1706	423755	4	30
:1782	424657	4	3
:1756	425516	4	0
:1779	427960	4	15
:1780	428044	4	14
:1755	460639	4	1
:1100	481734	4	8
:1122	488727	4	34
:1799	499269	4	10
:1800	499550	4	9
:1826	501672	4	48
:1823	501784	4	47
:1824	501708	4	46
:1825	501683	4	45
:1837	506828	4	23
:1838	507179	4	22
:1781	563066	4	3
:1892	630101	4	8
:1889	629971	4	4
:1890	630130	4	1
:1891	630155	4	0

Total query evaluation time = 776991 microsecs

Data on Results Return

QUERY - firstcousin(x y)?

20 PE/5 Bus Configuration (original version)

TIMES FOR RETURN OF RESULTS

Proc.No.	Time result found	Size of Packet	Pe.No.
693	191839	4	9
746	194203	4	13
747	196494	4	14
694	196546	4	8
745	199071	4	16
781	202897	4	11
782	203097	4	10
811	203402	4	4
812	203492	4	18
657	208683	4	5
658	210676	4	2
659	216403	4	19
634	216364	4	17
813	337543	4	16
985	341198	4	0
986	342441	4	7
633	346207	4	18
1030	352411	4	9
1029	353476	4	1
660	354956	4	19
748	362597	4	14
944	363446	4	6
1067	372914	4	8
1045	377732	4	10
1178	401456	4	11
1177	401813	4	12
1162	408705	4	3
1205	417231	4	13
1243	452738	4	5
583	482219	4	2
584	484661	4	18
1001	486244	4	0
1002	489744	4	7
1276	502377	4	15
1274	510222	4	9
1309	523495	4	19
1310	524951	4	10
1331	526771	4	1
1068	538030	4	6
1046	542055	4	8
1161	544327	4	11
1206	553827	4	3
1244	563627	4	4
814	615041	4	16
1273	629214	4	17
945	637986	4	0
1447	652255	4	2
1448	654162	4	18
1332	659782	4	19
1470	668246	4	15
1275	671433	4	9
920	687263	4	6
1514	725207	4	1
1513	725180	4	8
1562	758874	4	12
943	779871	4	7
1694	785898	4	4
1691	790899	4	11
1693	792644	4	16
1692	793041	4	17
1711	804371	4	3
1469	805382	4	5
1639	843016	4	13
1602	846886	4	6
1815	863455	4	19
1816	863872	4	18
1601	870129	4	8
1640	896999	4	12
1799	921645	4	0
919	929246	4	7
1712	940336	4	17
1800	1006292	4	19
1559	1012846	4	4
1914	1021169	4	10
1913	1023567	4	2
1911	1025858	4	6
1912	1025776	4	3
1560	1148791	4	4
946	1190143	4	0
1561	1284933	4	4

Total query evaluation time = 1290924 microsecs

Data on Results Return

QUERY -firstcousin(x y)?

20 PE/2 Bus Configuration (original version)

TIMES FOR RETURN OF RESULTS

Proc.No.	Time result found	Size of Packet	Pe.No.
561	189579	4	12
562	189677	4	11
724	204120	4	3
787	206092	4	19
788	207160	4	16
759	208633	4	8
760	208746	4	7
723	211713	4	4
672	214715	4	18
671	216112	4	0
803	232967	4	9
804	235976	4	1
858	281760	4	5
923	294416	4	2
922	294588	4	13
921	295153	4	17
876	302241	4	10
725	349514	4	7
979	364316	4	19
980	366475	4	18
1029	370905	4	12
1030	371189	4	11
1045	392517	4	4
546	393602	4	16
1119	422273	4	1
1120	425287	4	8
545	429578	4	0
1046	432032	4	15
875	439479	4	13
857	439986	4	5
855	451359	4	17
924	458449	4	10
1183	483085	4	2
726	501093	4	7
1133	502700	4	18
942	520391	4	12
941	520745	4	11
1134	529831	4	16
1121	558599	4	1
1122	563756	4	8
856	584518	4	13
1243	613419	4	6
1184	619291	4	2
1287	657617	4	10
1288	657706	4	3
1359	693070	4	18
1360	693325	4	14
1428	713786	4	19
1427	715083	4	0
1429	717366	4	11
1491	737677	4	9
1492	739249	4	4
1584	779124	4	12
1582	787860	4	8
1557	794378	4	6
1558	794626	4	5
1691	817945	4	15
1692	817990	4	13
1693	819382	4	10
1694	825968	4	3
1750	858552	4	2
1816	867383	4	1
1749	876788	4	7
1848	886790	4	0
1865	892529	4	18
1866	896032	4	9
1846	901827	4	4
1430	923936	4	11
1581	926507	4	12
1799	958475	4	15
1800	993300	4	14
1815	994730	4	2
1881	1003097	4	13
1882	1003328	4	10
1847	1018817	4	17
1845	1036546	4	9
1244	1042368	4	4
1583	1062612	4	12
1909	1150255	4	7
1910	1150598	4	6

Total query evaluation time = 1150598 microsecs

Data on Results Return

G7. Summary

The results included in this appendix are summarised in Chapters 8 and 9 in Figures 8.1, 8.2, 8.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.11 and 9.12. These chapter's also present additional data on function calling overheads and the analysis of the results.

Appendix H

PLL Program for "AND" Queries

```
/* Note that the following definitions contain no alternatives;
they have been written in this manner to test the performance of
the parallel system in the absence of process spawning */

define query0(x) tobe b(x) and (some(y)(c(x y) and d(y) and e(x y)))?
define b(x) tobe w(x) and (some(y)(v(x y) and u(y)))?
define e(x y) tobe s(x y) and r(x) and t(y)?
define t(y) tobe f(y) and (not(k(y)))?
define v(x y) tobe o(x) and (some(z)(l(z) and p(x z y) and q(z y)))?
define query1(x y z) tobe (x=((8*8)+(6*(sqrt(9))))) and
        (y=(z+(2*x)+(5*(sqrt(4))))) and (z=(9+10+(sqrt(16)*2*3)))?
define query2(x y z a b c) tobe (x=9) and (y=1) and (z=2) and
    (a=99) and (b=2) and (c=8) and s2(x) and t2(x) and r2(y)?
define query3(x y z) tobe a3(x) and b3(y) and c3(z) and d3(x z) and
    e3(x y) and f3(y z) and g3(x y z) and h3(z x) and k3(x) and
    l3(y) and m3(z) and n3(x y z)?
define query4(x) tobe a4(x) and b4(x) and c4(x)?
```

Appendix I

C Program for Measuring Function Calling Overheads

```
#include <timer.h>

/************************************************************/
/***********          Function Definitions          *****************/
/************************************************************/

/** The first group are designed to measure the overhead involved in
increasing the number of formal parameters in a function definition; the
second group look at the relationship between void and returning functions
**/

void  func12(val1,val2,val3,val4,val5,val6,val7,val8,val9,val10,val11, val12)
 int  val1,val2,val3,val4,val5,val6,val7,val8,val9,val10,val11,val12;
{;
}

void func6(val1,val2,val3,val4,val5,val6)
 int val1,val2,val3,val4,val5,val6;
{;
}

void func5(val1,val2,val3,val4,val5)
 int val1,val2,val3,val4,val5;
{;
}

void func4(val1,val2,val3,val4)
 int val1,val2,val3,val4;
{;
}

void func3(val1,val2,val3)
 int val1,val2,val3;
{;
}
```

```
void func2(val1,val2)
 int val1,val2;
 {;
 }

void func1(val1)
 int val1;
 {;
 }

void func0()
 {;
 }

int func2r(val1,val2)
 int val1,val2;
 {return(val2);
 }

int func1r(val1)
 int val1;
 {return(val1);
 }

int func0r()
 {return(34567);
 }

/**********************************************************/
/*******              Main Program              ************/
/**********************************************************/

main()

{int i;
 int time1,time2;
 int no; int no1=80876, no2=7865;
 int loop_count = 100000;
```

```
printf("Function timing starts\n");

 time1=timer_now();
 for (i=0;i<loop_count;i++)
  {func0();
 }
 time2=timer_now();
 printf("Time for 100,000 iterations of func0 =%d ms\n",
                                              (time2-time1)/1000);


 time1=timer_now();
 for (i=0;i<loop_count;i++)
 {func1(no1);
 }
 time2=timer_now();
 printf("Time for 100,000 iterations of func1= %d ms\n",
                                              (time2-time1)/1000);


 time1=timer_now();
 for (i=0;i<loop_count;i++)
 {func2(no1,no2);
 }
 time2=timer_now();
 printf("Time for 100,000 iterations of func2= %d ms\n",
                                              (time2-time1)/1000);


time1=timer_now();
for (i=0;i<loop_count;i++)
{func3(no1,no2,no1);
}
 time2=timer_now();
 printf("Time for 100,000 iterations of func3= %d ms\n",
                                              (time2-time1)/1000);
```

```
timel=timer_now();
for (i=0;i<loop_count;i++)
{func4(no1,no2,no1,no2);
}
time2=timer_now();
printf("Time for 100,000 iterations of func4= %d ms\n",
                                                  (time2-time1)/1000);


timel=timer_now();
for (i=0;i<loop_count;i++)
{func5(no1,no2,no1,no2,no1);
}
time2=timer_now();
printf("Time for 100,000 iterations of func5= %d ms\n",
                                                  (time2-time1)/1000);


timel=timer_now();
for (i=0;i<loop_count;i++)
{func6(no1,no2,no1,no2,no1,no2);
}
time2=timer_now();
printf("Time for 100,000 iterations of func6= %d ms\n",
                                                  (time2-time1)/1000);


timel=timer_now();
for (i=0;i<loop_count;i++)
 {func12(no1,no2,no1,no2,no1,no2,no1,no2,no1,no2,no1,no2);
}
time2=timer_now();
printf("Time for 100,000 iterations of func12= %d ms\n",
                                                  (time2-time1)/1000);
```

```
time1=timer_now();
 for (i=0;i<loop_count;i++)
 {func2r(no1,no2);
 }
 time2=timer_now();
 printf("Time for 100,000 iterations of func2r= %d ms\n",
                                        (time2-time1)/1000);


time1=timer_now();
 for (i=0;i<loop_count;i++)
 {func0r();
 }
 time2=timer_now();
 printf("Time for 100,000 iterations of func0r= %d ms\n",
                                        (time2-time1)/1000);


time1=timer_now();
 for (i=0;i<loop_count;i++)
 {no=func0r();
 }
 time2=timer_now();
 printf("Time for 100,000 iterations of func0r with assignment
                              =%d ms\n", (time2-time1)/1000);


time1=timer_now();
 for (i=0;i<loop_count;i++)
 {func1r(no1);
 }
 time2=timer_now();
 printf("Time for 100,000 iterations of func1r = %d ms\n",
                                        (time2-time1)/1000);
```

```
time1=timer_now();
 for (i=0;i<loop_count;i++)
 {no=func1r(no1);
}
 time2=timer_now();
printf("Time for 100,000 iterations of func1r with assignment
                                        =%dms\n",(time2-time1)/1000);

 time1=timer_now();
for (i=0;i<loop_count;i++)
 {func2r(no1,no2);
}
 time2=timer_now();
printf("Time for 100,000 iterations of func2r= %d ms\n",
                                                (time2-time1)/1000);


 time1=timer_now();
for (i=0;i<loop_count;i++)
 {;
 }
 time2=timer_now();
printf("Time for 100,000 null operations = %d ms\n",
                                                (time2-time1)/1000);


 time1=timer_now();
for (i=0;i<loop_count;i++)
 {no1=no2;
 }
 time2=timer_now();
printf("Time for 100,000 assignment operations = %d ms\n",
                                                (time2-time1)/1000);
```

```
 time1=timer_now();
 for (i=0;i<loop_count;i++)
  {no1++;
  }
 time2=timer_now();
printf("100,000 iterations of i++ = %d ms\n",
                                        (time2-time1)/1000);
}
```

Appendix J

The 3L Parallel C System on the Transputer

J1. Introduction

This appendix details the Transputer system used in the latter part of the project. The first section briefly describes the Transputer chip, and indicates how it has been used for the simulation. The second section describes the relevant part of the 3L Parallel C system and discusses the issue of software configuration.

J2. The Transputer

The Transputer is a specialised chip which has been designed to support parallel processing by the formation of interconnected networks of Transputers [INMOS 89]. The chip consists of three types of functional unit: a CPU, a small amount of RAM (typically 1 or 2 Kbytes) and four link units which control the communications channels. Each link implements a bidirectional communication path and can be used to connect with another Transputer thus enabling them to be connected to form various network topologies. The software model for programming this network is based on the concept of Communicating Sequential Processes: by dividing the computational task into modules that can be run on separate Transputers and defining the messages between modules to match the point to point communication channels, the network of Transputers can function as a parallel multiprocessor machine [Hoare 78], [Rentenerghem 89]. However the Transputer system used in the project consisted of a single chip; essentially the Transputer acted as a standard sequential processor unit and none of the facilities for parallelism were utilised. The reason for transferring the software to the Transputer system was to obtain the benefit of the fine granularity clock which was not available on the Sun workstation.

The outline of the Transputer system used in the project is shown in Fig.J1. This represents the relationship between the hardware components and the software modules needed to run the system. The B004 Transputer board consisted of a single T414b-15 Transputer and 2 Megabytes of external RAM; the Transputer ran at a speed of 15 MHz and held 2 Kbytes of on-chip

RAM. A Tandon PCA acted as host computer, the connection channels being implemented by one of the four Transputer links.

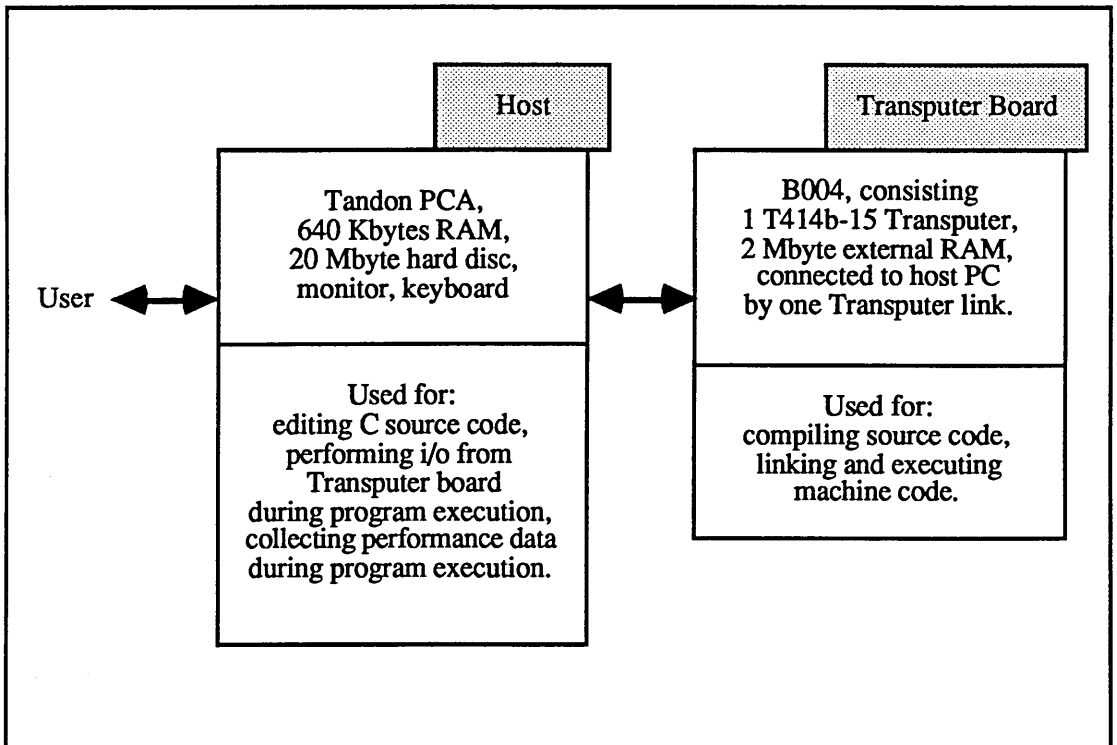


Fig. J1 - The Transputer System

J3. The 3L Parallel C System.

The standard programming language for Transputer systems is Occam which was developed by the manufacturers, Inmos [INMOS 88]. This allows the CSP model to be implemented by defining parallel processes which communicate by means of messages and channels. Within a process sequential algorithms are coded using conventional imperative constructs. Thus Occam includes standard high level language features and the facility to express parallel activity and message passing.

The 3L Parallel C system allows programs to be developed using the C programming language for the Transputer [3L C 88]. In the same way that Occam involves both parallel and sequential control constructs, Parallel C uses standard sequential C code to implement imperative algorithms within processes and additional "parallel" syntax to define communication and parallel activity. However these additional parallel features have not been used for the simulation software and are not considered here. For the single Transputer two parts of the 3L Parallel C system are of relevance: the software which is executed on the host computer to implement the interface between the user and the Transputer, and the Transputer software which is responsible for the preparation and execution of the user's program.

The simulation code was transferred from the Sun workstation to the host machine and amended to include the clock timing functions available within Parallel C. This alteration of source code was carried out using a standard editor on the host PC. The code was then compiled and linked on the Transputer using the 3L Parallel C compiler and linker, data being transferred from the host hard disc by means of one of the Transputer links. The system which was responsible for the execution of simulation involved two additional Parallel C software modules as well as the simulation module: a server program on the host machine and a filter process on the Transputer. The module "afserver" ran on the host machine throughout program execution and was responsible for channelling i/o messages to and from the Transputer by means of one of the links. On the Transputer the "filter" process was installed to act as an intermediary between the simulation software and the afserver program.

In order to obtain meaningful results from the simulation system care was needed in configuring the operation of the modules in the Transputer. The appropriate configuration was achieved by running a "config" process on the Transputer after linking and before execution of the simulation code. Two aspects of the system configuration were involved in this task: the relationship of the "filter" and simulation modules, and the memory management for the simulation code.

Because the filter task was running concurrently with the simulation module on the Transputer it was necessary to ensure that the simulation task was not subject to "hidden" delays because of interruptions by the filter process as this could have affected the timings obtained from the simulation. This was the first aspect of the configuration of the system that required attention. The Parallel C system was configured to give the simulation module high priority which ensured it would not be interrupted by the filter process except when i/o was necessary. When requests for i/o were made from the simulation software these were dealt with by the filter process and clearly it was important to ensure that the positioning of calls to the timer within the C code did no include these operations.

The second aspect of the system that needed configuration was the usage of on-chip and external RAM. By default the Parallel C system places as much of the C system stack in on-chip RAM in order to produce good performance from the system. However for most substantial programs the 2 Kbytes of on-chip memory is not sufficient for the total stack requirements and overflow into external RAM occurs. This arrangement means that absolute timings for parts of a program may vary with the state of the system stack: a function that is called at one point during program execution may take very much longer to complete than it does at an other stage in the program because the CPU is addressing values in off-chip memory. Measurements of these differences is shown in Fig.J2 which gives respective timing values for functions using internal and external RAM as the system stack. The functions used are those specified in Appendix I which were defined initially for determining the effect of parameters on function timings (see Chapter 9.3.1).

Function	On-Chip	Off-Chip
Null operation	185	283
Assignment operation	270	365
func0	440	635
func1	480	697
func2	529	753
func3	540	850
func4	635	968
func5	627	1057
func6	675	1181
func12	864	1728

Fig. J2 - Times in ms for 100,000 Iterations

The simulation software had been written to obtain data on aspects of the performance of the parallel PLL interpreter and real times were measured. When the disparity of times produced by the difference in "on-chip execution" and "off-chip execution" was recognised and understood it was decided that good comparative results could only be obtained by ensuring that only external RAM was allowed. The disabling of on-chip RAM could not be done directly on the T414b Transputer: the method employed was to use the 3L Parallel C configuration facilities to place dummy code on the on-chip RAM. By ensuring that this area of memory was occupied by code that was never used it was possible to obtained proper comparative measurements of different parts of the simulation code, although much of the performance advantage that is expected from the Transputer architecture was lost.