Technical Report

UCAM-CL-TR-216
ISSN 1476-2986

Number 216

Exploiting OR-parallelism in Prolog using multiple sequential machines

Carole Susan Klein

ISSN 1476-2986

Abstract

If the branches at each node of a tree are labelled, paths through the tree can be represented by a sequence of labels called an *oracle*. If an oracle leading to a node is followed, all of the bindings and other state information associated with the node will be recreated. Thus, oracles are both a specification for a path through the tree and a concise format for representing the environment at a particular node.

This dissertation investigates the use of oracles for the parallel execution of Prolog programs. The execution of a Prolog program can be represented pictorially by an AND/OR tree. The branches of OR nodes within this tree have no binding dependencies so their evaluation can be performed on separate processors. If one or more of these OR branches is explored in parallel, *OR-parallelism* is exploited in the Prolog program.

A distributed system called the *Delphi Machine* has been designed and implemented to exploit the OR-parallelism inherent in Prolog programs. In the implementation described in this dissertation, Delphi runs on a group of uniprocessors connected by Ethernet. Various *control strategies* using oracles to control the parallel search are investigated. The execution times for Prolog programs run on the Delphi machine are compared with those of a compiled and an interpreted sequential Prolog system. The results show that a distributed system using oracles to control the parallel search can be an efficient way to exploit OR-parallelism in nondeterministic programs.

Because of overheads imposed by the Delphi algorithm, a program executed on a single processor Delphi machine runs at approximately one half the speed as the same program executed on the unmodified Prolog system. For a twenty processor configuration, the speed ups obtained vary from approximately two to nine times depending on the amount of OR-parallelism which can be exploited by Delphi. Problems with large amounts of OR-parallelism show a nearly linear speed up.

Contents

Appendices

Chapter 1 Introduction

1.1 Branch Enumeration

If the branches at each node of a tree are enumerated, then the location of any node within the tree can be identified by a unique sequence of numbers. This sequence of numbers is called an *oracle* [Clocksin and Alshawi 1988]. At each node of the tree in Figure 1.1, the branches are enumerated from left to right. The oracle [4, 1, 2, 1] leads to the internal node indicated. An oracle is therefore a path leading to a particular location within a search space.

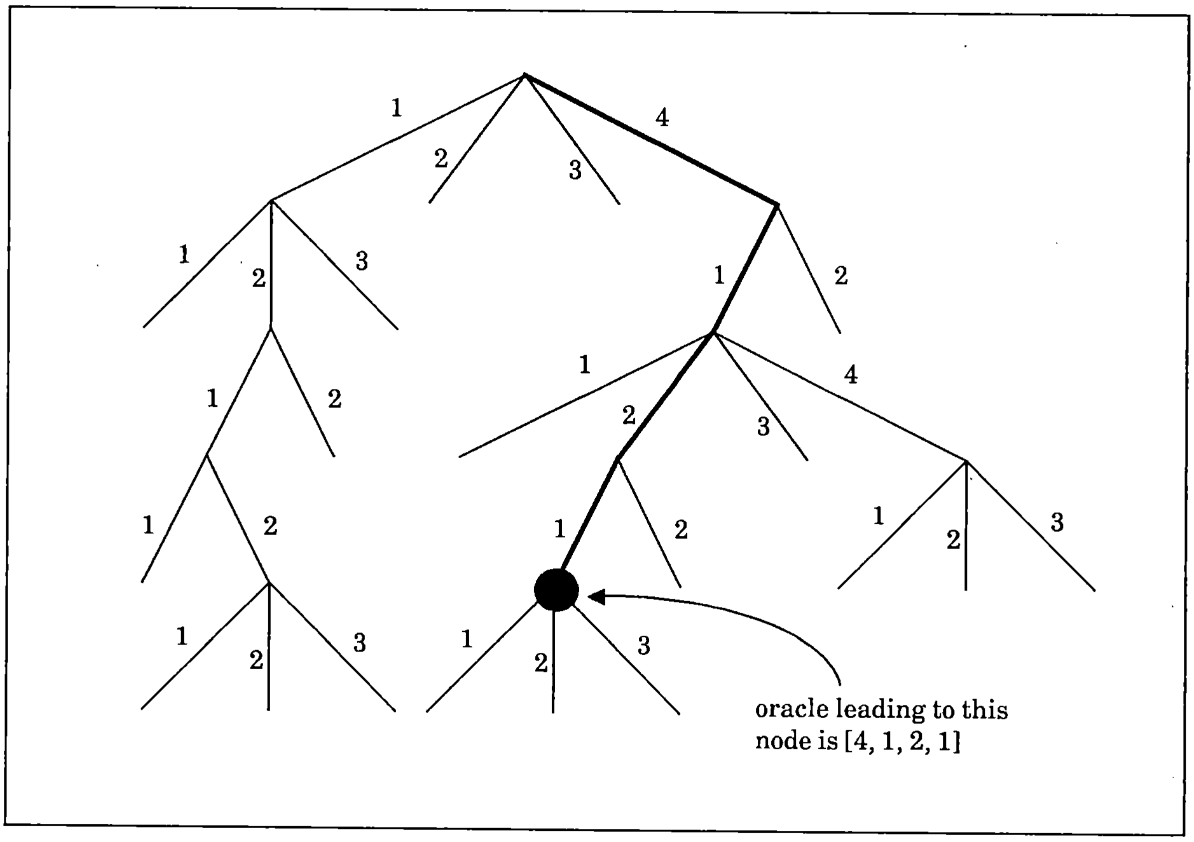


Figure 1.1 Branch Enumeration

An oracle can be *followed* by picking the clause at each node which corresponds to the next number in the oracle sequence. By following an oracle, the environment associated with each node is recreated. In Figure 1.1, as the oracle [4, 1, 2, 1] is followed, the environment associated with each of the nodes along the path is recreated. All of the bindings and other state information associated with each node will be generated. Thus, oracles are both a specification for a path through the tree and a concise format for representing the environment at a particular node.

1.2 Oracle Types

Figure 1.2 is the same example tree showing the location of solutions. Table 1.1 lists a few of the different types of oracles with example oracles corresponding to the tree in Figure 1.2.

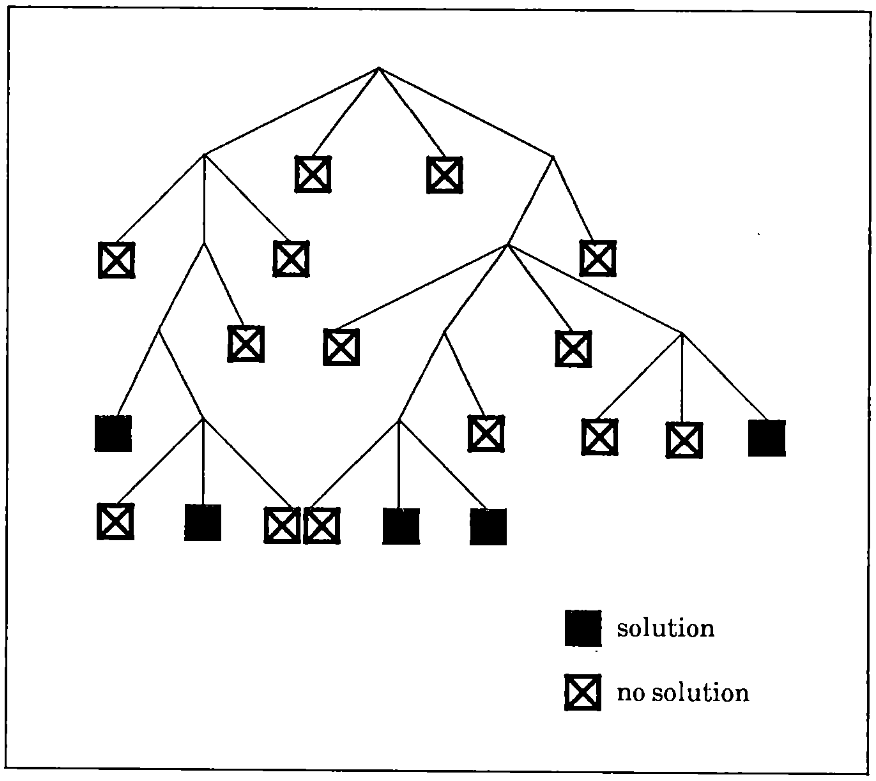


Figure 1.2 Example Tree Showing the Location of Solutions

1.3 Delphi Machine

One of the uses of oracles is in exploiting the OR-parallelism inherent in Prolog programs. Executing a Prolog program generates an AND/OR tree. The branches of the OR nodes within this tree can be partitioned among multiple processors without the need to communicate bindings. Each processor will perform the computation of following an independent path starting at the top of the search space. The binding information will be recreated from the root so there is no need to share bindings or communicate among the processors to ensure consistency. Paths through the search space can be explored in parallel by multiple sequential machines simultaneously following separate oracles. These host machines each execute an equivalent process (or processes) called the Prolog system.

Type	Description	Examples
incomplete	leads to an internal node	[1, 2] [4] [4, 1, 2, 1]
successful	leads to a leaf node with a solution	[1, 2, 1, 1] [4, 1, 4, 3]
failed	leads to a leaf node with no solution	[2] [1, 3] [4, 1, 2, 1, 1]
complete	reaches a leaf node and may have extra unused branch numbers	[3] [3, 1] [3, 1, 4] [4, 1, 2, 1, 2] [4, 1, 2, 1, 2, 3]
redundant	a complete oracle with extraneous branch numbers	[3, 1] [3, 1, 4] [4, 1, 2, 1, 2, 3]

Table 1.1 Types of Oracles

Since an oracle specifies a path starting from the root of the search space, it can be given to any Prolog system which:

- Knows how to follow an oracle.
- Contains the database of Prolog clauses for the program which is to be executed.

The oracle is an independent piece of information applied to a particular Prolog program. Multiple Prolog systems executing the same Prolog program can therefore be used to follow separate oracles simultaneously without the need to communicate with each other. This technique of giving different oracles to independent Prolog systems describes a parallel search of the tree and is the basis of the *Delphi Machine*.

The Delphi machine contains a Controller Process and any number of equivalent Prolog processes called *Path Processors*. In the original Delphi model [Clocksin and Alshawi 1988], the function of a Path Processor is to follow an oracle that it is given. It then reports back to the Controller the outcome of following that oracle. The outcome is:

- Incomplete if the oracle leads to an internal node.
- Successful if a leaf node with a solution is reached.
- Failed if a leaf node with no solution is encountered.

After an oracle has been followed and the outcome reported back to the Controller, the Path Processor is ready to receive a new oracle from the Controller. This cycle of receiving an oracle, following an oracle and reporting the outcome continues until the entire tree has been explored. The generation and distribution of oracles occurs according to a particular *control strategy*. The control strategies are algorithms which use oracles to partition the search space among multiple Path Processors and thereby explore the search space in parallel.

1.4 Implementation

Though an oracle is just a simple description of a path through a search space starting from the root of the tree, it is the fundamental implementation detail which is unique to the functioning of a Delphi machine. Throughout this dissertation, the word oracle is used in numerous contexts. It is the *job* that is sent to the Path Processors, or a string that is generated by the Controller, or the message that is received when a Prolog system performs a *check-in*. Each of these uses of an oracle depends upon the control strategy in operation. All of these uses end up equivalently being a means of pinpointing a node within the search tree by specifying a path to that location. Additionally, the oracle functions as a means of reestablishing the environment necessary to continue processing from the point where the path terminates.

The implementation of the Delphi machine described in this dissertation is similar to the original model proposed by Clocksin and Alshawi [1988]. A Controller process is used as the liaison between the Path Processors. The Path Processors run on independent host machines which are connected by a network. The original paper on the Delphi machine [Clocksin and Alshawi 1988] describes possible control strategies which can be divided into two categories; non-backtracking (where the Path Processors are not allowed to backtrack), and backtracking control strategies. The Delphi model described in that paper did not include an implementation.

The inefficiencies with non-backtracking strategies were discovered in the Delphi implementation by Alshawi and Moran [1988]. A few non-backtracking strategies were attempted in that research, but were found to be too inefficient for general use on a Delphi machine. Backtracking control strategies were the focus of the Alshawi and Moran implementation and are the most successful control strategies used on the Delphi machine described in this dissertation.

The implementation of the Delphi machine described in this dissertation was run on multiple uniprocessors connected by Ethernet. The host machines used for this research include five types of Digital Equipment Corporation hardware:

- VAXstation
- VAXstation II
- VAXstation II / GPX
- VAXstation 2000
- microVAX II

All of these are based on the same processor and have approximately the same processor speed. Throughout this document the term µVAX will be used to denote a single Digital Equipment Corporation machine listed above. The term µVAXes will be used for the plural.

1.5 Delphi and The Oracle

Delphi in ancient Greece was the site of the famous Oracle where prophecies were taken to be the will of the gods. Many Greek and Roman travellers came to the temple at Delphi to have important questions answered and their future revealed. The petitioner would go before a stone altar and make his request to the unseen woman behind. Her replies were often cryptic sayings which then needed interpretation by other priests associated with the temple. These cryptic replies were called oracles and today we refer to the place as the Delphic Oracle. In reality, the priestesses of the temple had many successes in fortune telling and hence the reputation of Delphi as a place for consultation flourished for over a thousand years [Parke and Worwell 1956].

One reason for this success was that Delphi was a major meeting point for travellers and the source of guidance to ambassadors from all over the world. The caretakers of the temple accumulated information brought to them from distant lands and artfully divulged it to those who sought fortune and fame. To the recipients of the oracle this knowledge was possible only through the clairvoyance of a god, when in fact, the temple often served as a database, accumulating information and dispensing it upon request. In return for this information the temple was lavished with gifts and treasures from all over the world, adding to the prosperity of Delphi and endowing its continued sponsorship. The Delphi machine is named after this sacred Oracle at Delphi.

1.6 Contributions of this Dissertation

This dissertation presents the first large-scale implementation of a multiprocessor Prolog machine based on the Delphi model. This Delphi implementation is a distributed system which contains facilities for process management, debugging and tracing, fault tolerance, and security.

Communications within Delphi are controlled through the use of two systems of Interprocess Communication (IPC) commands. Both the Amoeba-transactions-under-UNIX [Mullender 1987] and 4.2BSD [ULTRIX-32 Supplementary Documents: Volume III System Managers 1984] IPC systems were used in separate Delphi implementations. A detailed investigation of the problems encountered with these two IPC facilities is discussed along with some security implications.

A self-contained utility for managing multiple processes on multiple host machines was developed to organise and control the numerous Delphi machine processes. This *Process Management System* is a menu-driven tool which can assist in the debugging and control of distributed systems.

The process management facilities implemented for use with Delphi have extended the original model to cope with a multi-user environment. Multiple users can run simultaneous Delphi configurations over the network without interfering with each other. Additionally, a single user can run multiple Delphi configurations simultaneously. This provides a user the option of executing simultaneously separate Prolog programs on separate sets of host machines.

Numerous new control strategies have been developed such as: *branch by branch, automatic partitioning, reassign jobs, on demand* and *work estimate* strategies. The majority of the demonstrated results are by applications of automatic partitioning and the reassign jobs control strategies. *Automatic partitioning* allows the search space to be automatically partitioned without the need for the Path Processors to communicate. *Reassign jobs* is an extension of the automatic partitioning where idle Path Processors are given new partitions of the search space to explore. Adaptive control strategies such as the *on demand* strategy monitor the variation in parameters to allow the strategy to adjust dynamically to changing conditions. Different methods of encoding oracles for use by these control strategies is also discussed.

Results are shown for a variety of Prolog programs executed on the Delphi machine. These results are compared to both a compiled (Cosmic Prolog) and an interpreted (C-Prolog) Prolog system. Cosmic Prolog is the unmodified version of the Prolog system run by the Delphi machine. For a number of standard benchmarks, the characteristic patterns of communication between processors is also examined.

1.7 Chapter Discussion

Chapter 2 is a brief discussion of related work in the field of parallel Prologs. The major types of parallelism in Prolog programs (AND- and OR-parallelism) are initially defined. Some of the models which exploit these types of parallelism are then discussed. The Delphi machine was implemented to exploit the OR-parallelism in Prolog programs. Chapter 3 describes where this OR-parallelism comes from along with general methods for exploiting parallelism in search trees. A particular emphasis is placed on a description of how an AND/OR tree generated by a Prolog program can be transformed into an OR-only tree. Delphi extracts the OR-parallelism in Prolog programs by stacking the AND goals and partitioning the OR-only tree equivalent to the original AND/OR tree representation.

Starting with Chapter 4, the Delphi implementations designed during this research are investigated. Chapter 4 begins with the low-level details of *Prolog intermediate instructions* which provide the framework for exploiting OR-parallelism. A subset of the intermediate instructions called o*racle instructions* are the main source of oracle creation and manipulation by the Delphi machine. Chapter 5 and Chapter 6 examine the higher level aspects of the two Delphi machine implementations. Chapter 5 contains the details of communication commands while Chapter 6 explains how these commands are used in the control of the Delphi distributed system.

Controlling a parallel tree search involves the generation and distribution of oracles among multiple Path Processors. The algorithms which perform this function are called control strategies and are discussed in Chapter 7. Chapter 8 contains the results from benchmarking Prolog programs controlled by the strategies discussed in Chapter 7. Execution times for the Delphi machine are compared to those of two sequential Prolog systems. Further control strategies are discussed in Chapter 9. These strategies are extensions of the algorithms discussed in Chapter 7 to allow adaptive control. Some of the tools developed during the course of this research are discussed in Chapter 10. A few of these tools are useful both for the Delphi machine and for applications which are not associated with the Delphi research. Chapter 11 is a summary of what was accomplished by the Delphi research.

Chapter 2 Related Work

Many methods have been proposed for increasing the efficiency of Prolog by exploiting sources of parallelism inherent in the language. A brief description is given of some of the models which have motivated the Delphi research.

2.1 Definitions

The roots of parallel logic programming languages can be traced to a general interest in multiprocessor architectures [Hwang and Briggs 1984] and successes with exploiting the concurrency in imperative languages [Gehani and McGettrick 1988]. Kahn and MacQueen [1977] proposed a model of computation which formalised the description of process interaction within a parallel logic language. This process interaction model became the impetus for the development of stream-parallel implementations. Names have been given for various types of parallelism in a Prolog program by Conery and Kibler [1981]. The major interest in the field is focused on exploiting what is called AND- and OR-parallelism. Within these two types of parallelism, subdivisions have been identified. AND-parallelism is divided into restricted and stream AND-parallelism [Conery 1987], with types of OR-parallelism distinguished by *don't care* and *don't know* nondeterminism [Kowalski 1979]. Additional types of parallelism have been identified, and these can be grouped under the headings of AND- or OR-parallelism. These major categories are described below.

AND-parallelism	when more than one goal is contained within a clause, these goals can be processed in parallel.
restricted AND-parallelism	the goals to be processed in parallel do not have any shared variables in common.
stream AND-parallelism	goals which have variables in common can use the binding of a common variable as a synchronisation mechanism. One literal is said to be the producer of a binding while one or more other literals are consumers.
unification parallelism	concurrently unifying all of the terms within a goal with the terms in the head of a candidate clause.
OR-parallelism	when more than one alternate clause is defined for a relation, they can be processed in parallel.
don't care nondeterminism	some alternative clauses may not be tried.
don't know nondeterminism	all alternative clauses are tried, and all solutions to a query are found.

search parallelism

a parallel search is performed to find all of the clause heads which match the current goal.

Using these definitions, Delphi can be described as an OR-parallel model employing don't know nondeterminism. This type of nondeterminism is also known as full OR-parallelism as all alternative clauses are fully explored and all solutions are found.

2.2 Early Models

Implicit parallelism is the automatic identification and exploitation of sources of parallelism within a program. The idea is to maintain the syntax of a sequential Prolog language rather than adding special operators for exploiting parallelism. The alternative approach is to have explicit parallelism where the programmer must explicitly point out the portions of the program that are to be executed in parallel. Early languages [Clark and McCabe 1982, Porto 1982, Wise 1984] contained numerous programmer annotations to explicitly label parallel or sequential events. Delphi uses the implicit parallelism approach where the procedures involved with exploiting the parallelism in a program are transparent to the programmer. The syntax used in a Delphi program is standard Edinburgh syntax [Clocksin and Mellish 1981].

Early models of parallel Prolog-like systems demonstrated the difficulties in dealing with nondeterminism. To accomplish concurrency may mean sacrificing some of the distinguishing features of sequential Prolog. Synchronisation and process interaction must be included in the concurrent models, and in many cases, this involves limiting or obliterating sequential features which allow nondeterminism. One common restriction is to disallow the reversibility of a procedure's usage. New annotations must be added to the language to specify a *mode of use* of a procedure. These annotations are used to partially order the evaluation of goals, and so increase the efficiency of execution.

IC-Prolog [Clark and McCabe 1982] relies on variable annotations to provide predetermined producer and consumer relationships among dependent goals. In addition to these, numerous control annotations are supplied which order goals by delaying or forking a process. Epilog (Porto) [Porto 1982] and Prism [Kasif, Kohli and Minker 1983] provide annotations which allow the user to specify a sequence of goals to be evaluated sequentially or simultaneously. The ordering mechanism of Epilog includes weak and strict sequencing operators in addition to symbols that facilitate coroutines. The evaluation order is controlled by a complex mechanism which performs the reduction of goals one resolution at a time. Prism employs a simple notation of brackets and parentheses which delineates groups of goals to be solved asynchronously, and from those which must have a strict left-to-right ordering. Another Epilog (Wise) [Wise 1984, Wise 1986] uses various constructs to impose a sequencing order to allow both AND-parallelism and OR-parallelism. Optional variable annotations similar to those in IC-Prolog [Clark and McCabe 1982] are also provided. These take preference over the implied sequencing and help to order the goal evaluation.

Various methods for the synchronisation of goal reduction were borrowed from the longer established study of functional languages, and the formal models of parallelism [Hoare 1978, Milner 1980]. Mechanisms such as data triggered coroutines [Clark and McCabe 1982], thresholds [Wise 86], modes [Conery 1987], and read-only variables [Shapiro 1987b] are an attempt to control the data flow within a program and schedule the execution of goals in an efficient manner. Guards are another syntactic addition to parallel logic languages used both to restrict the nondeterminism in a program and to synchronise the execution of goals. The notion of a guard originated from Dijkstra's guarded commands language [Dijkstra 1976]. This concept has influenced the design of many parallel logic programming languages.

When the Fifth Generation Computer Systems Project was studying ways in which logic programming could be realised on a parallel architecture, the following approaches were considered [Fuchi and Furukawa 1983]:

(1) Addition of parallel control primitives to Prolog.
(2) Delaying the evaluation of goals until specific data arrives.
(3) Imposition of guards on Horn clauses and restriction of nondeterminism.

The third approach was chosen and so called *committed-choice* languages (those languages which use guards) became popular in the logic programming community.

2.3 Committed-Choice Languages

Concurrent Prolog [Clark and Gregory 1986], Parlog [Shapiro 1983], and Guarded Horn Clauses [Ueda 1985], are three very popular committed-choice languages. Their common ancestor, Relational Language [Clark and Gregory 1981], was proposed in 1981, the same year in which the Fifth Generation Computer Systems Project announced that logic programming was to have a central spot in their research. The *commit operator* is the common syntactic addition to the committed-choice languages. Its function is to restrict the search space of the program to increase efficiency in the language by *committing* to a particular nondeterministic choice. After this choice has been made the control mechanism does not permit backtracking to consider an alternative. A guarded clause has the form:

$$A \leftarrow G_1, G_2, \ldots, G_m \mid B_1, B_2, \ldots B_n \quad (m,n \geq 0)$$

The commit operator '|' is both a synchronisation and a control operator. Logically it functions just as a normal ',' operator (conjunction operator) would. That is, A is true if all of the G's and all of the B's are true. The order in which the goals are tried is not strictly left to right. Goals in the body of the clause (the B's), cannot be tried before the guard (the G's) have succeeded. Once the goals of a guard are determined to succeed the execution mechanism commits to this clause, and the execution of any alternative clauses is aborted. Forms of both AND- and OR-parallelism are used in this committing process. AND-parallelism is used when the guard goals are tried concurrently. If the guard succeeds the goals of the body can be concurrently tried.

An OR-parallel search is performed on the guards of the matched heads. It is only after the commit operator is passed in one of the clauses (meaning all the goals of a particular guard were successful), that the OR-parallel search is aborted and the chosen clause is *committed* to. This type of nondeterminism has been described as don't care nondeterminism.

Only one solution can be found with this don't-care nondeterministic method. If all solutions to a query are required, additional special purpose relations must be added to these languages. Committed-choice languages provide synchronisation methods which express logic programs as parallel systems of processes. They use shared variables in the form of streams as the communications medium. Their differences can be seen in the syntactic alterations on a standard Prolog program to order the evaluation of goals.

Read-only variables are the synchronisation mechanisms introduced in Concurrent Prolog [Shapiro 1983]. A new syntactic component is added to the language in the form of a postfix operator "?" which can appear after a term in the head or clause body. The unification algorithm is constrained by the additional requirements on a read-only term X?:

(1) If X? is unbound, then unification with a non-variable fails.

(2) If X? is unbound, then unification with a variable succeeds. The result of this unification will be a read-only variable.

(3) If X? is bound to u, unification with a term t succeeds if u and t are unifiable.

Binding a read-only variable to a value is a passive process; it must take place through the binding of the same variable in a non-read-only form somewhere else within the clause. The operational effect is that a process (in the form of a goal trying to reduce itself) is suspended until one or more of its read-only terms can be unified. The read-only annotation controls the data flow through shared variables and synchronises the processes.

Mode declarations are required for relations in Parlog [Clark and Gregory 1986] to constrain the communications among processes. If the declaration has an input constraint as one of its arguments, the unifier must supply an input substitution for that parameter. If the mode declaration shows an output constraint, the unifier must not supply a substitution. In this case, the calling procedure must have an unbound variable in that argument position. Suspension of a process occurs if unification would involve the violation of an input constraint. The caller in this case would not have had its parameter sufficiently instantiated. A run time error occurs if an output constraint would be broken by unification with a calling procedure. In Parlog, the procedure declaration determines its mode of use, and this never changes throughout the program. In Concurrent Prolog the caller of a procedure can partially affect its usage. If a caller has a read-only annotation on one of its variables, this constitutes an input declaration. A parameter which is to be in output mode only can be created by a read-only annotation on a variable in the head of a clause.

Guarded Horn Clauses [Ueda 1985] have no additional operators added to the language except for the commit operator. There is, however, a particular style of programming that is required for a query to be successful. All output binding information must be located in the body of a clause. Since the guard of a clause is not permitted to permanently instantiate any variables of its caller (this is true in all of the committed-choice languages), Guarded Horn Clauses suspends the computation of a guard until sufficient binding information is provided by the caller. This results in a guard being suspended until input arguments are bound. If the clause is selected for computation, unifications providing output bindings occur in the body. Evaluation of guards can be a very complex procedure. For this reason, the more efficient *flat* versions of Guarded Horn Clauses and Concurrent Prolog have been proposed [Shapiro 1986].

The trend for more recent committed-choice languages has been to introduce methods which allow multiple solutions to a query to be searched for in parallel. P-Prolog [Yang and Aiso 1986] incorporates a user controlled method which admits both don't-care and don't-know nondeterminism. The synchronisation of goal reduction is obtained by performing an *exclusive check* on the clauses which describe a procedure. Expected exclusive clauses (which are partitioned by the programmer) are evaluated with the following results:

(1) If none of the clauses in a relation are committable, the result is false.

(2) When the guards of more than one clause succeed, the calling process suspends.

(3) If only one clause is committable, the result is success.

In the case where more than one clause is committable and the clauses are annotated as non-exclusive, full OR-parallelism will be performed in an attempt to find multiple solutions to a query. For many of the older committed-choice languages a great deal of effort has been put into providing methods for acquiring all solutions to a query. The original design was to have two different subsets of the language. One *sublanguage* would be efficient and provided only a single solution to a query. An alternate sublanguage would supply all solutions to a query [Shapiro 1989]. Converting OR-parallelism into a more easily exploitable form is another method to allow multiple solutions to be found [Ueda 1986, Codish and Shapiro 1986].

Two additional committed-choice languages have been proposed which allow multiple solutions to be searched for in parallel. Saraswat [1987] provides a *don't know commit* operator to permit alternate clauses to be explored in parallel. Okumura and Matsumoto [1987] use the notion of layered streams where separate sections of the stream are available to separate processes.

2.4 OR-Parallel Models

Two major methodologies have come to light in the creation of OR-parallel Prolog models:

(1) Using a global address space (possibly with additional local memory) for the storage of multiple environments.

(2) Using separate processors as independent sequential machines.

When reduction of multiple clauses is attempted in parallel different bindings for variables may occur as the result of unification with the heads of more than one alternate clause. The traditional OR-parallel method involves the creation of OR-processes [Wise 1986]. Every relation that contains more than one alternative clause will have a child process spawned for each of these clauses consisting of a copy of the current bindings. The end result of this method is that a complete binding environment is created for every possible solution. If these multiple environments are independent, then storage space is at a premium and communication overheads are introduced within the copying time. A structure-sharing technique may minimise the amount of storage required. This may also increase communications among the processors when a new process traverses its ancestors' frames while searching for a value. Various OR-parallel algorithms have been proposed as an efficient method for maintaining multiple environments.

2.4.1 Global Address Space Designs

The single-assignment property of unification (once a variable is bound to a value it cannot be changed) ensures that processes created by OR-parallel search can share already bound variables with their parent process. The technique of sharing *committed contexts* reduces the time required by traditional OR-parallel techniques [Wise 1986] to copy environments. This is mainly because the entire address space of the shared environments do not need to be copied when execution is performed on a shared memory multiprocessor. Sharing of environments in this model is at the expense of a higher access time to establish a binding for a variable. In a proposed distributed binding model [Ciepielewski and Haridi 1983], environment directories are used which point to contexts/frames (groups of variable bindings often corresponding to a clause) which are either committed (have no unbound variables) or uncommitted (have at least one unbound variable). Committed contexts can be shared among the processes while uncommitted ones must be duplicated for each child process created. Values can be accessed in constant time by the two level storage system presented, but the entire directory of the parent process must be searched every time a new environment is constructed. An improved model is presented in [Ciepielewski and Haridi 1983] which links the directories into a tree structure and avoids the drawback of scanning the directories at the expense of a longer access time for the first reference to a context.

A variable importation algorithm [Lindstrom 1984] imports unbound variables from an ancestor into a local frame temporarily. Any unification will only bind variables which occur within the local frame consequently minimising the time required for updates. Variables are imported via an import mapping vector corresponding to the number of variables in a parent frame. References to unbound variables are pointers from the vector while bound values are nil. When a clause terminates, an export mapping vector is created to export variables back to their original locations in the parent frame. Any newly created variables which remain unbound are exported (via an export mapping vector) in an analogous fashion and a new parent frame is created.

The observation that only a small number of variables in ancestor frames are bound by unification with a clause head, prompted the idea of hash windows for variable updates [Borgwardt 1984]. A local frame is maintained for variables within the child process' clause and a hash window for any bindings that occur to variables referenced in an ancestor frame. Hashing on a variable's address provides a very fast look up mechanism for subsequent accesses and is efficient in terms of memory usage [Crammond 1985].

Two models of OR-parallel execution have been defined by Warren: the Naive model and the SRI model [Warren 1987a, Warren 1987b]. The Naive model is similar to the traditional OR-parallel model. A new process is spawned for each branching point in the search space. A process contains goals and a set of variable bindings. Bindings already created before the choice point is reached are added to the binding environment of the new process. An improvement in the efficiency of the SRI model [Warren 1987b] lead to the creation of the Naive model. This improvement ensured that access to variable bindings could be performed in constant time at the expense of some additional bookkeeping tasks.

Hash tables are also used for a certain type of variable in the Argonne Model of OR-parallel execution, which identifies three storage classifications for variables [Butler, Lusk, Olsan and Overbeek 1986, Overbeek *et al. 1985*, Shen 1986]. Private references correspond to variables which will not be shared by any other processor. This occurs when no additional OR-parallel nodes need to be explored since none exist below the current position in the search tree; program annotations in the form of compiler directives control the OR-parallelism. It is guaranteed that the value on the stack is the correct binding with a private reference, but multiple bindings may be possible with other variables. All other references contain a conditional flag. If the current path is the favoured one (left-most successful path) bindings on the stack will be correct, otherwise values must be looked up in a hash table.

All of the models discussed in this section on global address space differ from Delphi in the initial assumptions about implementation hardware. These models assume that a shared memory architecture is the target machine. The initial design of the Delphi model assumes that the processors involved will be independent machines connected by a network. This seems to be the more general case. Delphi could be implemented on a shared memory architecture with minimal difficulty, whereas a model designed for a shared memory may not be able to execute efficiently without it.

2.4.2 Sequential Prolog on Multiple Processors

An alternative to having a clever scheme for maintaining multiple environments is to make the environment for a goal or group of goals as local as possible to a particular processor. Each processing element (PE) would then be able to run standard sequential Prolog, and communication among PEs would only be necessary when a task makes a request to be split. Models designed to use sequential Prolog on multiple processors do not try to conserve storage, as separate environments are located on separate processors. They do attempt to minimise the quantity of job splits which in turn minimises the communications time among the processors.

One method extends sequential Prolog by splitting the control stack into process bundles (chunks of the search space) and allocating these portions to be searched by particular processors [Yashuhara and Nitadori 1984]. Partitioning of a task is performed in a demand-driven manner by an idle processor receiving a partition, recreating the environment and resuming processing. An algorithm which splits jobs in a depth first manner is described in [Sohma, Satoh, Kumon, Masuzawa, and Itashiki 1986]. As successive idle processors demand a job, the highest node in the search tree which has not been previously split is sectioned into two pieces. The interrupted processor continues its search in a depth-first left to right manner, while the demanding processor takes all the branches to the right.

Models described in this section are similar to Delphi in that they use a sequential Prolog running on multiple machines. A major difference is that the models described here take the active view that idle processors should demand jobs from other processors. This is one of the techniques explored by the Delphi research, however, the passive technique of having processors periodically check-in to see if there are any idle processors was more successful for the Delphi machine.

2.4.3 Broadcast Architectures

Hardware modifications have been proposed which reduce the time involved in copying an environment by implementing a broadcast architecture. When a job needs to be split, the working PE can broadcast a copy of its current environment to several idle PEs simultaneously and efficiently. If OR-parallelism was left unbridled, contention from burdened processors requesting a split of their workload or idle processors demanding a task may still occur. Various algorithms have been suggested that reduce the number of requests to split a job in an effort to reduce or avoid this contention.

In addition to the specialised architectures, some sort of split scheme algorithm is required to create an efficient system. The optimised solution is to avoid the overheads involved with splitting a job by employing a clever partitioning strategy which initially partitions the tree into fairly equal chunks. In this manner, the processors would be equally loaded throughout the computation, and no splits would be necessary. As this is impossible (since the shape of the search space cannot be known in advance), strategies are developed to minimise the splits as much as possible. In [Ali, Fahlen and Karlsson 1986] several split schemes are proposed. One of these is a depth-first iterative deepening strategy where each processor explores the search tree in a breadth-first manner to a particular level of the search space.

The splitting strategies in [Ali, Fahlen and Karlsson 1986] are similar to the automatic partitioning strategies of Delphi. The differences occur both in the hardware used to implement the system and the method of passing an environment to another processor. Delphi uses multiple sequential machines and [Ali, Fahlen and Karlsson 1986] uses specialised hardware with an architecture set up to allow broadcasting of environments. When an environment is passed, the information sent is the variable bindings and other information needed for control. With Delphi, the information sent is the concise oracle only.

The most important aspect of the research shown in [Ali, Fahlen and Karlsson 1986] is the use of local memory as far as possible so that binding environments do not need to be shared. It is only when idle processors exist and a job needs to be split, that any communications are performed. This is similar to Delphi where the environments must remain local to the processors as no shared memory is provided. This idea leads to the work shown in [Ali and Wong 1988] where the control strategies become very similar to ones proposed in [Clocksin and Alshawi 1988]. The major difference again is the hardware used for the implementation. A shared memory architecture is used in [Ali and Wong 1988]. With the shared memory comes the problems associated with contention, and various locking strategies have to be examined.

2.4.4 OR-Parallelism with AND-Parallelism

The AND/OR process model [Conery and Kibler 1985] and its modifications [Hermenegildo and Nasr 1986, Lin, Kumar, and Leung 1986], form a milestone in the literature by emphasising the trade-offs between the quantity of extractable parallelism and the combined effects of variable dependency analysis and program annotations. Static data analysis [Chang and Despain 1985] may miss some of the available AND-parallelism, as a worst case must be assumed while a complete run-time determination involves overheads due to the dynamic monitoring of variables.

Restricted AND-parallelism [DeGroot 1984] involves a compromise between complete compile time and complete run time data dependency analysis. In [Hermenegildo 1986], a more generalised notion of restricted AND-parallelism termed *goal independence* is formed. Goal independent AND-parallelism can be identified within several early implementations [Clark and McCabe 1982, Monteiro 1982, Kasif, Kohli and Minker 1983], and is still a prominent area of investigation.

Conery and Kibler [1985] detail the steps required to extract parallelism within their process model. An ordering algorithm organises goals within a clause on the basis of generator/consumer relationships among the variables. Parts of the algorithm are performed at compile time, parts are left for run time application. The connection rule is an heuristic used within the algorithm to aid in an optimal assignment of generator/consumer pairs. Forward execution is the graph reduction technique which coordinates the initiation of descendent processes. All of the above rules coupled with mode declarations are sufficient to implement AND-parallelism of mutually independent subgoals only for deterministic procedures. If the model is to be extended to include nondeterministic procedures an additional rule, backward execution, must be provided.

Sun and Tzu [1986] have proposed a combined OR-parallel/restricted AND-parallelism model of Prolog. A partitioning is performed such that goals within the same group have shared variables, and this grouping is mutually independent of all others. One special group is created containing goals which have a shared variable with at least one member of all the other groupings. Independent OR-trees are created from these groups and these collectively are termed the OR-forest. If some of the OR-trees can be searched in parallel then restricted AND-parallelism is accomplished.

2.4.5 Independent Processing via Clever Splitting Algorithms

Models of OR-parallel Prolog in this category avoid the necessity of a global address space by localising the computation of a particular branch in the search space. Instead of having processors allocated to a particular goal, all processors in the system start their execution by reducing the top-level (query) goals. This redundant execution continues until a choice point is reached, when a splitting algorithm is used to determine which processor(s) take which branch(s). After the split is performed and all branches are allocated, the processors continue just as a sequential Prolog interpreter would until the next choice point. These systems are dependent upon a clever splitting strategy which will ensure that no part of the search space is ignored (very important for all-solution queries), and attempt to divide the workload efficiently.

The Multi-Sequential Machine (MSM) proposed by Ali [1987] is designed to partition branches of the search space onto different processors so that no communications are involved, yet each processor knows what branch(es) to take at any choice point. This is done with a set of local pointers which relay information concerning the number of processors currently executing a particular branch, and the virtual number of each processor. A right-biased strategy works as follows:

> All processors begin at the root of the tree until the first choice point. At this point, one processor (PE1) takes the leftmost branch and continues as a standard sequential Prolog. The other processors (PE2-PEn), continue redundant processing of the tree until another choice point is reached when PE2 now takes the leftmost branch. This continues until either the complete search space has been explored (which meant that there were more processors than independent paths from the root to a leaf node), or all processors have been allocated a task and the highest number processor (PEn) has to explore the remainder of the right part of the search space on its own.

MSM also allows the copying of environments to take place after a certain number of processors have become idle. A working processor is interrupted to request a portion of its environment, and this is copied to the idle processors simultaneously through a broadcast link. To implement the copying process, a manager process is required in addition to the multiple sequential machines.

Both the MSM and BC machine [Ali Fahlen and Karlsson 1986] proposals describe a balanced strategy for partitioning the search space:

> All processors begin execution at the root of the tree. When a choice point is reached, the branching factor is determined and the processors are equally divided among the branches.

Many of the Delphi strategies described in this dissertation can be considered to fall within the category of independent processing via clever splitting strategies. Automatic partitioning is a method of automatically splitting up the search space among independent processors. The reassign jobs strategy is an extension of automatic partitioning where the idle processors are given new areas of the search space to explore.

2.5 Using Oracles

The distinguishing feature of Delphi type work is the use of oracles to define a position in the search space of a Prolog program. Though they may not be called oracles, the concept is the same for other models which use an enumeration technique for distinguishing between multiple choices. A path is created consisting of the choices to take at each branching point. A choice is just a number specifying which branch to take. In the seminal work on Delphi machine, Clocksin and Alshawi [1988] define an oracle and show how it can be used to specify paths within the search space. Alshawi and Moran [1988] implement a Delphi machine and demonstrate its successful performance on a parsing problem. Other researchers have started to consider the idea of using an oracle as a concise method of representing either an environment or a path to a particular location in the search tree.

Shapiro [1989] shows an algorithm and implementation of OR-parallelism into Flat Concurrent Prolog which makes use of paths through the search space. These paths are a list of the indices of the clauses to be explored, hence they are oracles. In [Wang 1989] an oracle is termed a refutation path. The refutation paths are used to control the unification of literals. Oracles have been used in OR-parallel systems such as the implementations described in this dissertation and in [Clocksin 1987, Clocksin and Alshawi 1988, Alshawi and Moran 1988]. Oracles have been used for implementing the OR-parallel component of an AND-parallel Prolog [Shapiro 1989]. Oracles have been investigated for use in exploiting unification parallelism [Wang 1989]. Research into the use of oracles to exploit AND-parallelism is underway at the University of Cambridge Computer Laboratory [Wrench 1989].

Chapter 3 Focus on OR-Parallelism

OR-parallelism is demonstrated in a graphical manner by showing how an OR-only tree can be created from an AND/OR tree. With an OR-only tree, the sites for exploiting OR-parallelism are every internal node with a branching factor greater than one.

3.1 Sequential Prolog

The following example Prolog program is taken from Clocksin [1987].

```
g(U, V)   :-  p(U), q(V), r(U, V).
p(1).
p(2).
q(1).
q(2).
r(X, X).
```

with goal clause `:- g(X, Y).`

The search tree corresponding to the execution of this program is shown in Figure 3.1. The AND node is denoted by an arc drawn across the branches to its descendants.

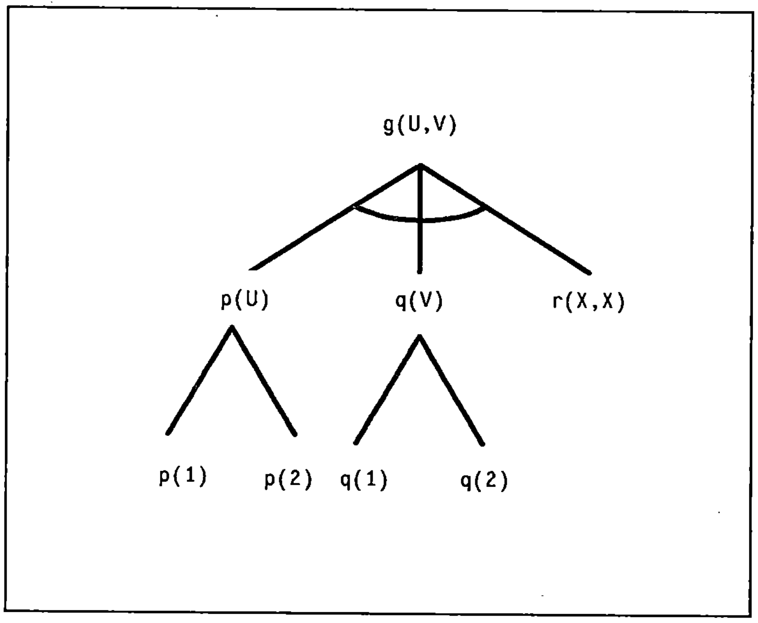


Figure 3.1 Example AND/OR Tree

Figure 3.2 shows how a sequential Prolog system would search the example tree. The top-level goal clause `g(U, V)` is invoked matching the first and only predicate named 'g' with arity two (`g/2`). Matching the head of clause `g/2` causes an AND stack to be created which holds the three subgoals contained in the body (`p(U), q(V), r(U, V)`). Figure 3.2 continues the search from this point with the exploration progressing down the page.

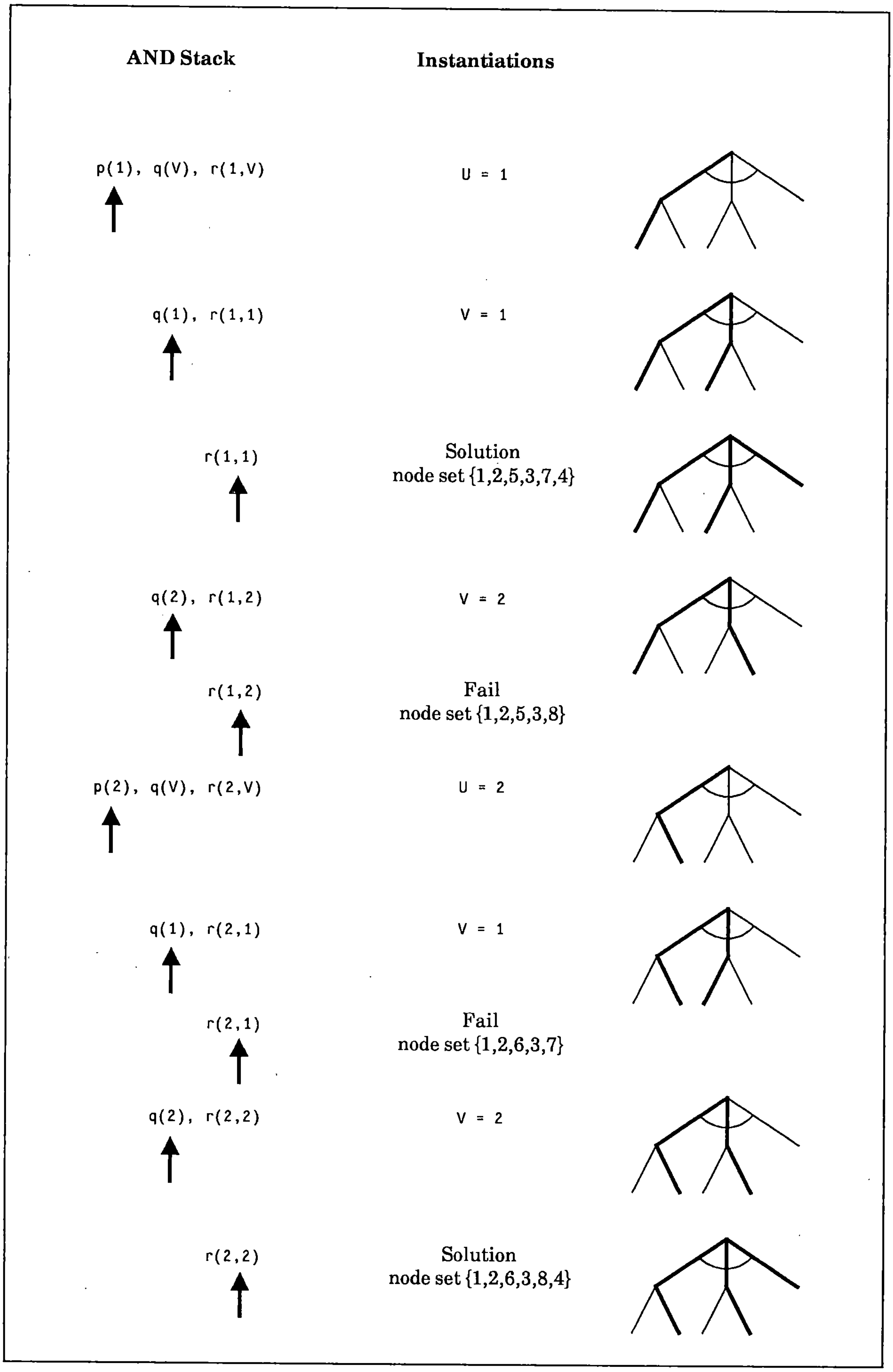


Figure 3.2 Sequential Prolog Search

The arrow in the AND stack column points to the top of the AND stack, with the subgoals to the right of the arrow contained further down in the stack. When a solution has been found, an implicit failure is assumed which forces backtracking and continues the search.

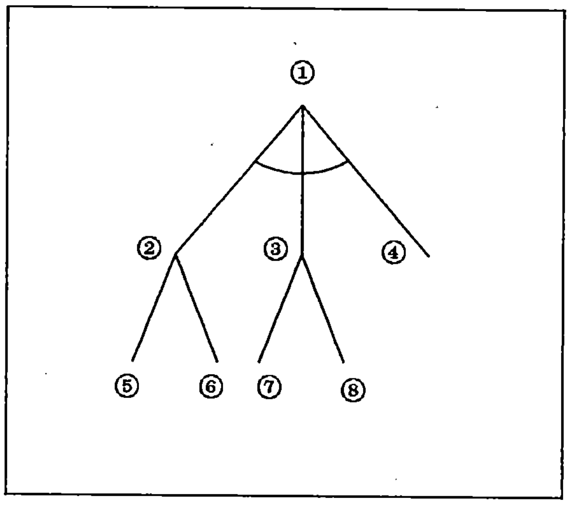


Figure 3.3 AND/OR Tree Node Labelling

Labelling the nodes in the tree from top to bottom, and from left to right we get Figure 3.3. Exploration of the search tree involves four sets of nodes corresponding to the four combinations of OR branches in the tree. Sequential Prolog creates these combinations by exploring the four node sets in the following order:

1,2,5,3,7,4
1,2,5,3,8
1,2,6,3,7
1,2,6,3,8,4

With sequential Prolog the entire search space is explored by a single processor, so the tree has been partitioned into one single large chunk. Next we consider how to partition the search space into multiple sections to exploit either AND or OR-parallelism. After a tree has been partitioned, a parallel search is executed with each partition being explored by a separate processor.

3.2 AND/OR Partitionings

Partitioning the tree in any manner would allow more than a single processor to search the tree simultaneously. The dotted lines in Figure 3.4 represent the boundaries of separate partitions of the tree. This tree is split by two vertical lines partitioning the tree into three sections. The first section contains nodes {2, 5, 6} the second section contains nodes {1, 3, 7, 8} and the third contains node {4}. If we had three processors to search this tree, it would be very convenient to have them search in parallel with Processor 1 exploring the nodes in set 1, Processor 2 exploring set 2, and Processor 3, set 3. This type of parallelism is AND-parallelism as each processor is receiving one of the branches of the AND node to explore. The difficulties with AND-parallelism is that there may be shared variables among the branches of the AND node. When there are variables shared among the branches, and these branches are explored on separate processors, then the processors must somehow communicate their bindings to assure consistency. With OR-parallelism there are no shared variables and the separate branches of an OR node can be explored by separate processors without the need to communicate bindings. If there were more or less than three processors available to explore the AND node, then a *splitting algorithm* would control which processors were assigned to which branch of the AND node. In many cases this splitting algorithm would be biased to favour one section of the AND node over the others.

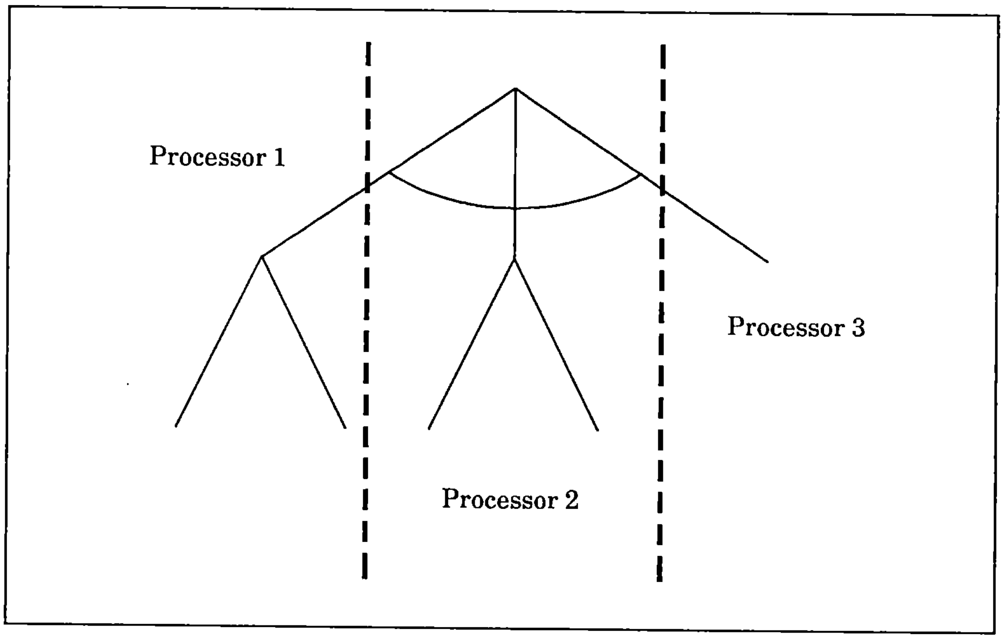


Figure 3.4 AND Partitioning

The biasing applies to which side of the tree is to be favoured. This is a guess that more of the work will be located on that side of the tree. If there is more work on a particular side then either more processors should be assigned to explore that section of the tree or, that side of the tree should be contained in a smaller partition. Figure 3.5 shows both of these types of bias. In Figure 3.5a two processors are splitting the AND node with a left-hand bias. The assumption is that there will be more work to perform on the left-hand side of the tree. The left-most branch is placed in a partition all by itself to be explored by a single processor or equivalently, Processing Element (PE). Figure 3.5c shows this left biasing when there are four processors available to split the AND node. Two of the processors are assigned to the left-most branch (PE1 and PE2), and the other two PEs are each given a single branch. When there are more PEs than branches, more than one PE is assigned to the branches on the side of the tree which the splitting bias is applied to.

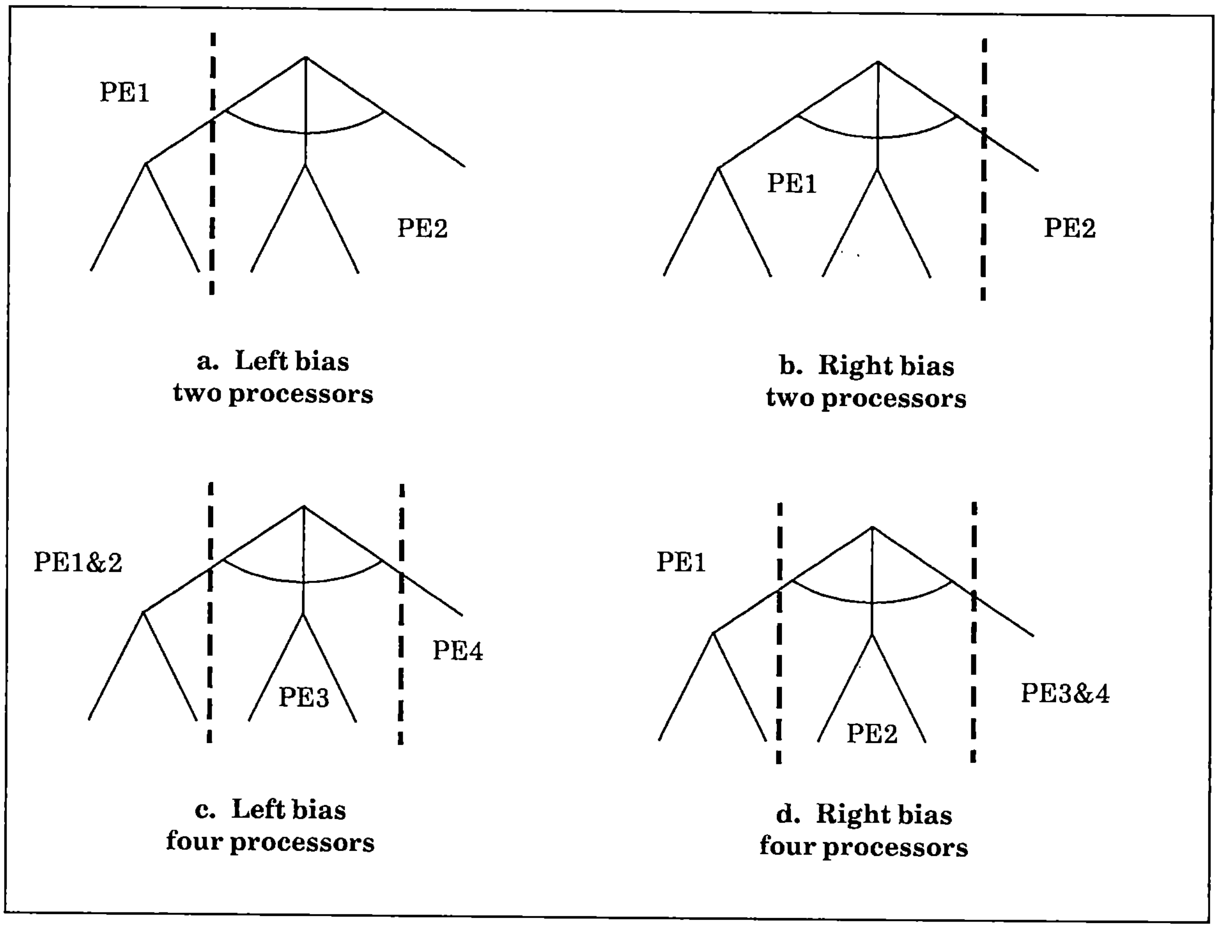


Figure 3.5 Biased Partitions

In all of the cases so far, the branches of the AND node have been split among multiple PEs without splitting any of the OR nodes. OR nodes can be split among a number of processors so that each of the branches of the OR node are tried simultaneously. Figure 3.6 is an example of splitting an OR node between two processors. PE1 explores the left branch of the OR node, and PE2 explores the right branch.

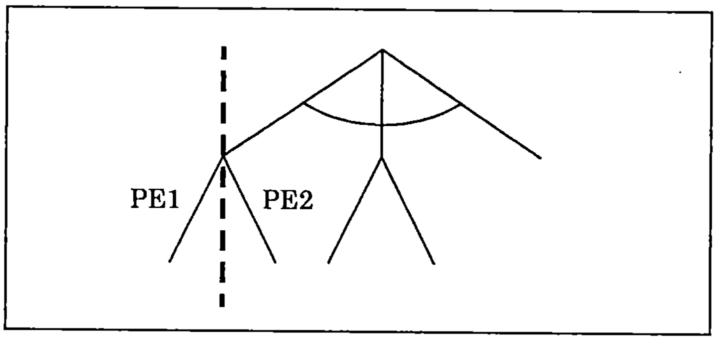


Figure 3.6 Splitting an OR node

This is a very inefficient partitioning since node 5 (see Figure 3.3) is explored by PE1 only, node 6 is explored by PE2 only, with all of the other nodes in the tree explored by both PEs. The sequence of traversal for each of the PEs is performed simultaneously resulting in the following node sets being explored:

Processing Element 1	explores nodes	1, 2, 5, 3, 7, 8, 4
Processing Element 2	explores nodes	1, 2, 6, 3, 7, 8, 4

This shows that there is duplication of effort in searching most of the tree. Duplication of work is not necessarily an undesirable part of partitioning the search space. If we consider that nodes 5 and 6 may not be the leaves of the tree, but may have very large subtrees below them, then the duplication becomes less important. With very large subtrees below nodes 5 and 6, partitioning only the single OR node could reduce the time for searching the entire tree by nearly half. Figure 3.7 shows how two processors would partition this new tree. The work of exploring the two very large subtrees below the branches of the OR node is split between the two PEs.

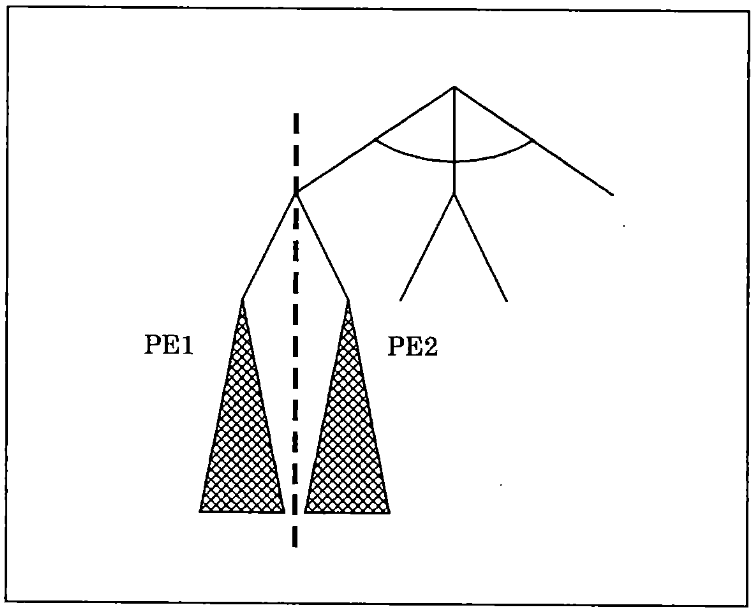


Figure 3.7 Partitioning Two Large Subtrees

It is just as simple to partition each of the OR nodes in this search tree, and have each of the processors automatically pick the correct branch to take. Using the same tree as an example, assume that two processors are exploring from the root of the tree. At each OR node, a processor takes the branch number corresponding to that processor's identity number. Assume that the branches in the OR node are numbered from left to right. Figure 3.8 shows the result of having two PEs search the tree.

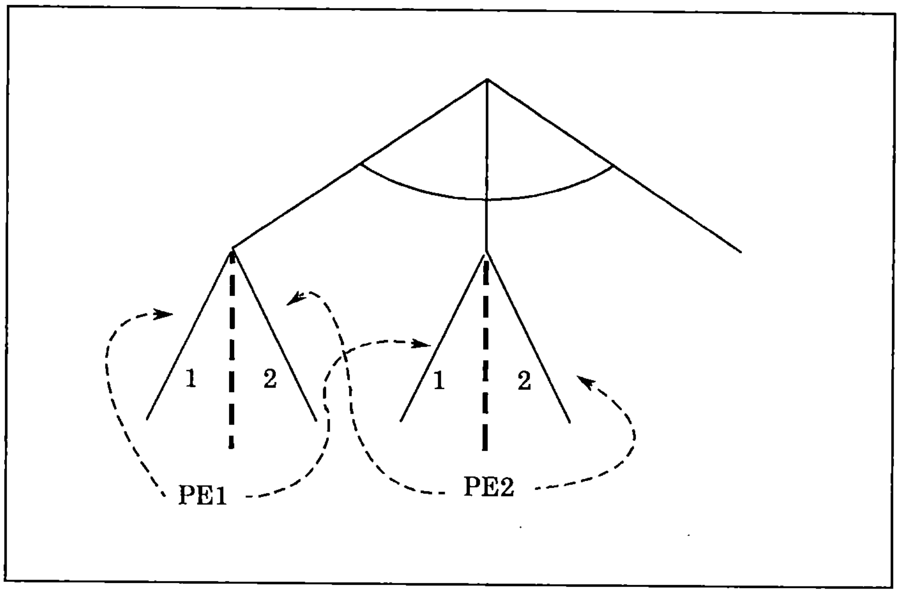


Figure 3.8 Incorrect Partitioning

Whenever an OR node is explored, each of the two processors takes the branch number corresponding to that processor's number. Processor 1 always takes the first branch and Processor 2 always takes the second. In this manner, there is no duplication of the OR node branches. In this strategy the PEs simultaneously explore:

Processing Element 1	explores nodes	1, 2, 5, 3, 7, 4
Processing Element 1	explores nodes	1, 2, 6, 3, 8, 4

This is an incorrect partitioning (not a complete search of the tree) since not all combinations of OR branches have been explored. The two sets shown here are equivalent to the first and fourth node sets explored by sequential Prolog (see Figure 3.2). There are two sets missing in this search:

(1) The combination containing the first branch of node 2 with the second branch of node 3.

(2) The combination containing the second branch of node 2 with the first branch of node 3.

A simpler way to look at the searches performed by sequential Prolog and Delphi is to transform the AND/OR tree generated by Prolog into an OR-only tree.

3.3 OR-Only Trees

For Delphi to be able to exploit the OR-parallelism in an AND/OR tree, the 'AND' goals are stacked and then sequentially executed, just as they are with sequential Prolog. For explaining search strategies it is useful to have a pictorial representation of an AND/OR tree in its transformed state of an OR-only tree. This is done by duplicating some of the branches of an AND node and omitting some other AND branches. Throughout this dissertation the partitioning of OR-only trees among multiple processors is shown. The AND nodes are still being executed by the

Path Processors (for now, a Path Processor can be thought of as equivalent to a PE), but as they are executed sequentially, they are not shown in most of the diagrams. Delphi only exploits the OR-parallelism (or OR branches) of and AND/OR tree, so the AND nodes are generally not shown. This section demonstrates how an AND/OR tree can be transformed into an OR-only tree. Figure 3.9 shows the same example tree being rearranged so that the OR-parallelism is clearly shown.

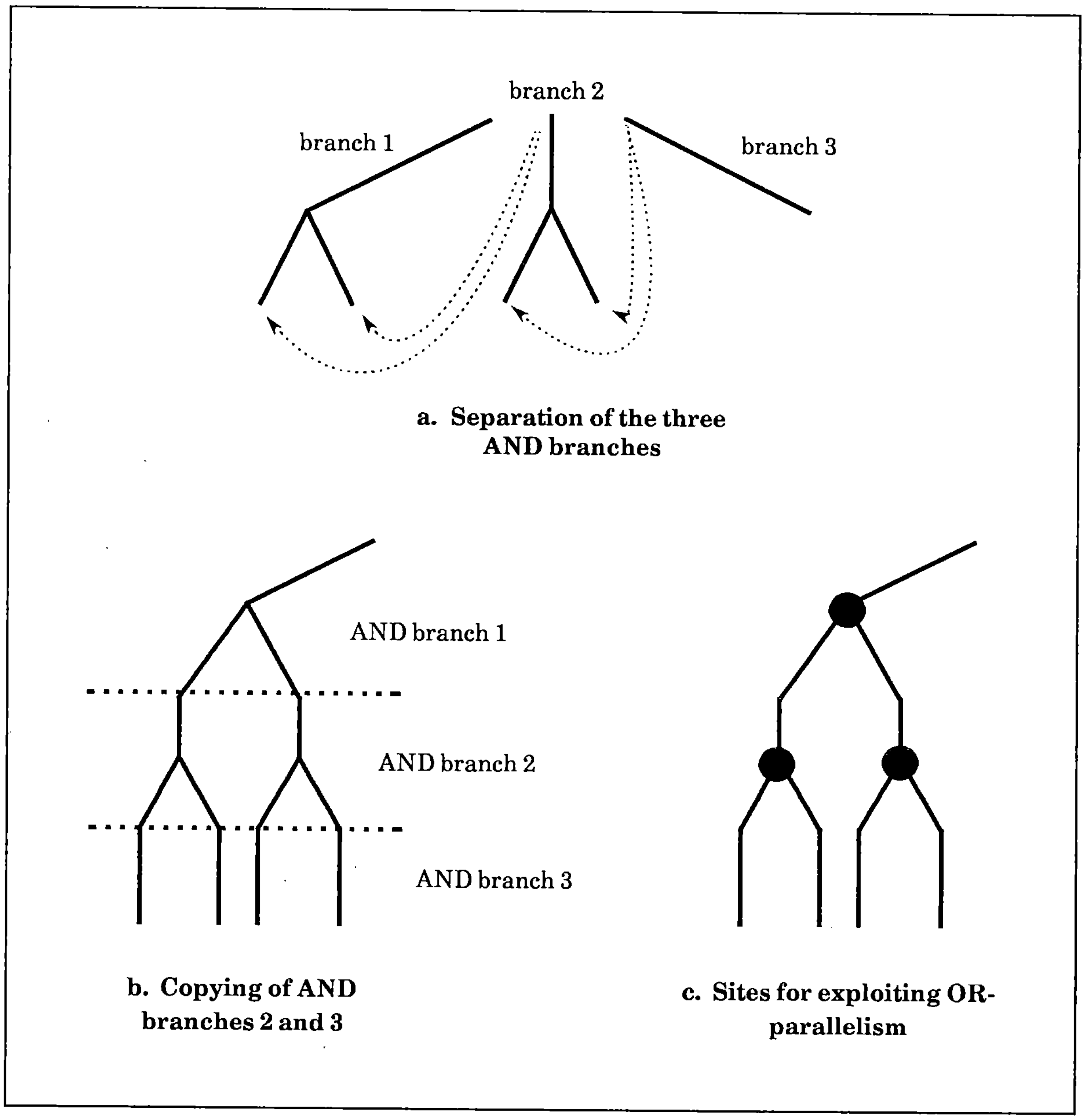

branch 2
branch 1
branch 3
a. Separation of the three AND branches
AND branch 1
AND branch 2
AND branch 3
b. Copying of AND branches 2 and 3
c. Sites for exploiting OR-parallelism

Figure 3.9 Creating an OR-only Tree

Every branching point on the OR-only tree is a potential site for exploiting OR-parallelism. All of the AND nodes have been removed, so there will be no problems with maintaining consistency of bindings across partitions. Given this form of the original tree, it is easy to demonstrate how two processors would perform an exhaustive search. They would both start at the root of the tree and then split up at the first OR node or *choice point*. After this split, they would each continue to search the tree in the manner of sequential Prolog (depth-first left-to-right search with backtracking) stopping back at the node where the Processors originally split up. Labelling the nodes in this OR-only tree we get Figure 3.10.

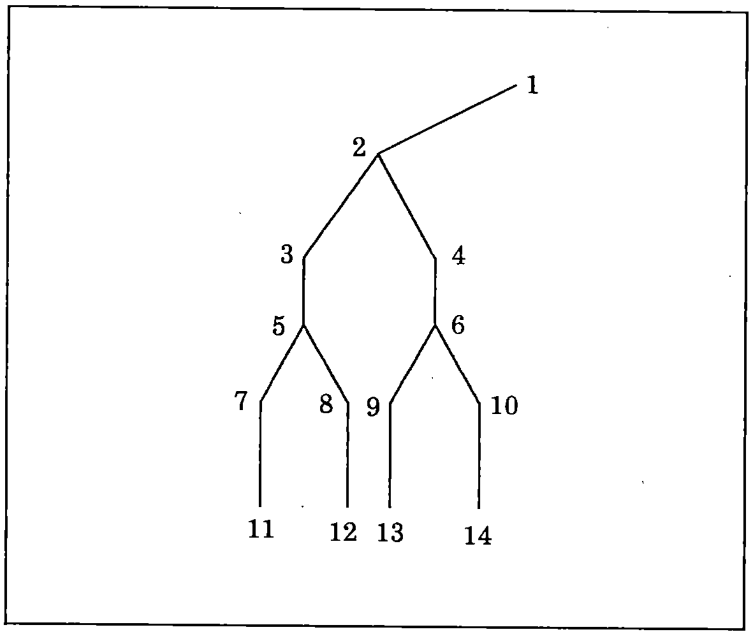


Figure 3.10 Labelling Nodes in the OR-Only Tree

Standard sequential Prolog searches the nodes of this OR-only tree in the following order:

1, 2, 3, 5, 7, 11, 8, 12, 4, 6, 9, 13, 10, 14

This search is called a depth first left to right search with backtracking. In this search there is no partitioning of the tree, therefore the search can be accomplished by a single uniprocessor. Also, since there is a finite number of branches, and each of these branches is a finite length, this tree can be completely searched by the single processor.

Two Processing Elements could simultaneously search the nodes of this OR-only tree:

Processing Element 1	explores nodes	1, 2, 3, 5, 7, 11, 8, 12
Processing Element 2	explores nodes	1, 2, 4, 6, 9, 13, 10, 14

For use in describing Delphi searches, this OR-only tree still contains redundant information. There are seven deterministic OR nodes (and OR node where the outward branching factor is equal to one) in this tree. These branches do not represent true choices so are not displayed in most of the OR-only trees within this dissertation. The proper OR-only tree corresponding to the original example search space is shown in Figure 3.11. All of the non-leaf nodes in this tree are sites for exploiting OR-parallelism.

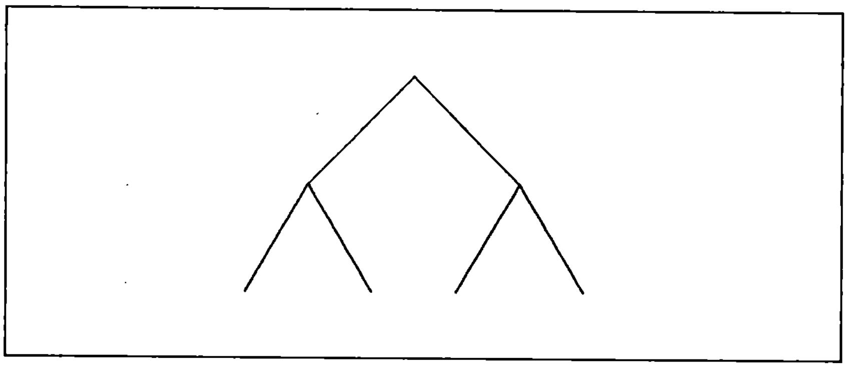


Figure 3.11 OR-Only Tree without Deterministic Branches

Figure 3.12 shows how the OR-only tree without deterministic branches can be partitioned by multiple Processing Elements. Figure 3.12a shows a partitioning for two PEs searching the tree. PE1 and PE2 both start at the root of the tree, and then split up at the first choice point. Below that point, the processors explore the entire subtree as if they were a single sequential Prolog. If there were four processors available, they could partition the work as shown in Figure 3.12b. All four PEs start at the root of the search space. PE1 and PE2 take the left branch at the first choice point. PE3 and PE4 take the right branch at the first choice point. This same type of split occurs at the second choice point reached by each of the PEs.

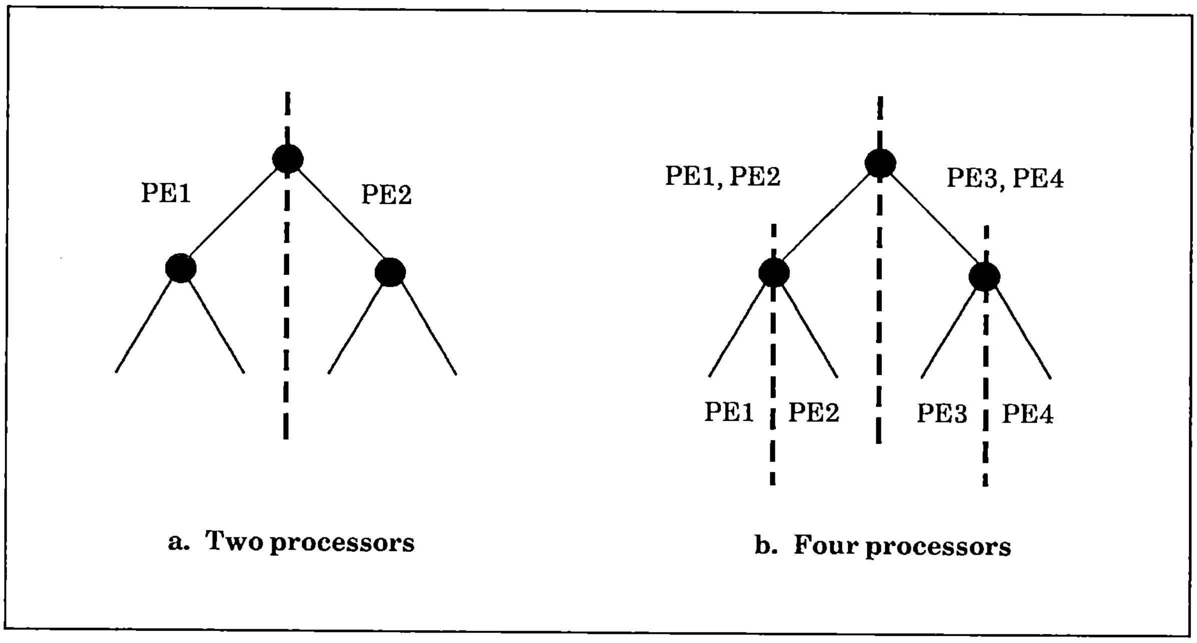


Figure 3.12 Exploiting OR-parallelism

In a Delphi search, the execution of AND branches (of a particular AND node) is duplicated on each of the Path Processors which encounters the AND node. The creation of an OR-only tree from the AND/OR tree shows this duplication of AND branches. The tree itself however is not actually generated or held in memory. The clause instructions to be executed can be found only once in memory, and they are held sequentially. These instructions are not duplicated in the *program space* which contains the compiled Prolog program. The compiled code is stored in the same way as in a sequential Prolog system. Chapter 4 goes into more detail about the program space maintained by each of the Path Processors and the intermediate code corresponding to a source Prolog program.

Chapter 4 Oracle Instructions

This chapter describes the intermediate instructions for the creation and manipulation of oracles. The addition of oracle instructions to the WAM (Warren Abstract Machine) [Warren 1983] allows OR-parallel search within the control strategies of Delphi. Paths are communicated to and from the Controller process by using the oracle number associated with each clause in a *set*. This oracle number is a parameter of what are called `onum` or oracle numbered instructions. An instruction named `setmax` relays information that a choice point has been reached before any choice is made. Instruction `setmax` has as its argument the number of clauses in a set; this is equivalent to the number of choices which must be tried. Nine oracle instructions make up the full complement of intermediate code needed for implementing any of the Delphi control strategies. These instructions allow OR-parallelism to be exploited and turn the WAM into a WAMO (Warren Abstract Machine with Oracles).

4.1 Delphi Prolog

A source Prolog program goes through the following three stages of the Delphi system:

(1) Delphi Prolog Compiler
(2) Delphi Loader
(3) Run time Prolog System

The Delphi compiler takes source Prolog code and generates an intermediate code similar to WAM instructions. Special indexing instructions for all clauses in a predicate are also generated by the Delphi compiler. The Delphi loader takes the compiled code and generates oracle instructions from the output indexing information while loading the program for execution. The run time Prolog system consists of the code which executes the WAMO intermediate instructions in addition to the facilities needed for communication with the Controller.

The compiler is a separate component of Delphi which is run on a single host machine. The compiled code is then distributed to all of the Path Processors along with the system files needed to execute the intermediate code. The Path Processors contain both the Delphi loader and the run time Prolog system which load and execute the compiled program according to the current control strategy.

4.2 Delphi Compiler

At the beginning of the Delphi research, two Prolog systems were candidates for modification; SICStus (Swedish Institute of Computer Science) Prolog, and SB-Prolog (Stony Brook Prolog, from the State University of New York at Stony Brook). SICStus was originally chosen and modified to perform simple oracle manipulations. This system was soon rejected due to the large amount of code contained in the version that we were using. SICStus had many additional features which were not needed by a Delphi implementation, and the smaller public-domain SB-Prolog was investigated.

SB-Prolog incorporates a WAM-style compiler written mostly in Prolog with an interface to the C programming language for various low-level components. The compiler produces a symbol table with offsets which are resolved by the loader, individual clause information given on a predicate-by-predicate basis and information used by the loader for the generation of indexing instructions. Only the indexing methods of the original compiler were significantly changed for the Delphi implementation. Since oracles are intimately involved with the indexing procedures it is important that this portion of the code be as clear and maintainable as possible.

Information given by the compiler which is used to produce a symbol table include:

- Symbols such as predicate names and constants.
- The type of each symbol.
- A relative entry point if the symbol is a predicate name.

Predicate names along with their arity (number of arguments) are put into the symbol table as each new predicate is compiled. The bulk of the work performed by the compiler is in the creation of clause instructions which are the low-level instructions to perform the job of matching against a goal pattern. Each of the clauses within the predicate generate numerous WAM instructions in an attempt to unify each of the arguments in a clause head and maintain the proper arguments in the available registers. The first piece of information generated for each clause is the entry point to that clause's instructions. The entry points to the predicates and clause instructions are used when the oracle instructions are created.

Indexing information has been added to the original SB-Prolog compiler to create the Delphi Compiler. Every clause in a predicate has some indexing information associated with it. The information is given for each clause in the order in which it appears in the program. The format of this information is:

```
<type>  <value>  <entry point>
```

type is a single character indicating the type of the first argument in a clause
value is a four byte field with various meanings
entry point is a four byte address which is the entry point to a clause

There are seven characters which represent seven types of first argument:

n	nil
l	list
c	constant
s	structure
i	integer
v	variable
z	zero arguments (for predicates with zero arity)

The value field may contain two different pieces of information depending on the type of the first argument. For the constant and structure types, value is a pointer to that object's name in the symbol table. With an integer, value refers to the numeric value of that integer. The value field is used for hashing multiple clauses with the same type of first argument which differ textually. For example, clauses within the same predicate may have either a '1' or a '5' as their first argument. These clauses will all be of type integer, but the integers they represent are different. These clauses would be hashed according to the value of their integer first argument. For constants and structures, the same principle applies. The constant or structure arguments can be subdivided into categories containing an exact textual string of a particular constant or structure. Clauses with either nil, list, variable, or zero as its type do not require a value to be associated with them. Each of these types represents only a single category of argument, so there is no need to hash them. The value field for the types nil, list, variable or zero will be null.

4.3 Delphi Loader

An example of intermediate instructions created by the compiler and loader is shown in Appendix 4a. A source Prolog program is given along with its loaded format to demonstrate how indexing is performed by the Delphi loader. The appendix contains examples of most of the oracle instructions which are described in this chapter.

Indexing information emitted by the Delphi compiler is used by the Delphi loader. Oracle instructions are generated at load time and arranged in a uniform format after the symbol table and clause instructions have been loaded. The entry point of a predicate becomes the first available memory location after the symbol table and clause instructions. The instruction name for the entry point of any predicate is jumponspecial. Instruction jumponspecial takes five operands or arguments which are pointers leading to each of the possibilities for any type of first argument in the goal which is *special.*

The first argument of a goal to be matched is either special or general. A general argument is a variable with a special argument being anything other than a variable. Zero arity goals do not have a first argument so are neither special nor general. There are five categories of special argument: nil, list, constant structure, or integer. The jumponspecial instruction has five pointers as arguments associated with each category or set. If the first argument of a goal to be matched is special, then one of the five pointers is followed. A jump to a special set is taken if the first argument of the goal is special. If the first argument of the goal is general, then no jump is taken and the program pointer falls through to the category or set of clauses which is located directly below the jumponspecial instruction.

The format for indexing instructions is equivalent for all predicates with arity greater than zero. The entry point of these predicates is the jumponspecial instruction which has pointers to the five types of special set. If the goal clause does not have a special first argument, then a fall through occurs to the code directly beneath the `jumponspecial` instruction. For zero arity predicates, the `jumponspecial` instruction is not needed. Since there are no arguments to index, the goal will match the heads of all candidate clauses in the predicate. For this reason, the zero arity predicates have a different format for their entry point and indexing instructions. There is no `jumponspecial` instruction as the first instruction to the entry point. The entry point to the predicate leads to the first `onum` instruction of a single set containing all clauses within the predicate.

Clause instructions generated by the compiler, are located before the entry point to the predicate. The predicate entry point begins with a `jumponspecial` instruction which has five arguments associated with it. These five arguments are pointers to sets of clauses whose first arguments will most probably match the given goal.

4.4 Oracle Numbered Instructions

Modifications to the original SB-Prolog WAM include the addition of instructions named 'oracle numbered instructions'. These instructions contain an argument which describes that clauses position within a group or set of clauses. Since this argument is a numeric field, these instructions are called `onum` instructions where `onum` is an abbreviation for oracle number. There are four `onum` instructions used by the Delphi Prolog system to index and execute clauses:

● `onumtry`	build a choice point
● `onumretry`	update a choice point
● `onumtrust`	remove a choice point
● `onumsing`	do not build a choice point

Three of these four new instructions are analogous to their WAM counterparts try, retry and trust. The fourth (`onumsing`) is a special instruction for Delphi where you want a Path Processor to pick some clause and then never backtrack to it again. This instruction is often used in the execution of non-backtracking control strategies. All four `onum` instructions have the same format. This format contains an instruction name followed by three arguments:

```
onum<suffix>     arg1     arg2     arg3
```

The `onum` instructions each refer to an individual clause within a predicate. `Arg1` is the arity of the predicate and therefore the arity of the clause which is being referenced. `Arg2` is the number of this clause within the set, and `arg3` is the entry point for this clause. As an example, consider a set which has four clauses in it and has been loaded to run as part of a non-backtracking strategy:

`onumtry`	2	1	entry point for clause number 1
`onumretry`	2	2	entry point for clause number 2
`onumretry`	2	3	entry point for clause number 3
`onumtrust`	2	4	entry point for clause number 4

The first argument to each onum instruction is the integer 2 showing that the arity of the predicate (of which these clauses are a part) is 2. The second argument is the *oracle number* of the clause. This number represents a clause's position within a set of clauses which have been indexed together. The entry point to each clause's low-level instructions is given by the third argument. Each clause will have the same arity number as all others in the set, but will always have a unique oracle number and a unique address describing its entry point.

Onumsing has the semantics that a choice point is not created before executing the clause specified in the third argument. It is not the same as an onumtrust instruction which says to relinquish the last choice point on the choice point stack. There are two places where onumsing is used. The first case is where there is only one clause within a set to choose from. The other case is when a non-backtracking control strategy is being used. With non-backtracking strategies, all of the onum instructions will be onumsing instructions. Since no backtracking is going to be performed there is no need to create any choice points to backtrack to. With non-backtracking control strategies, onumsing ensures that no choice points are ever created or destroyed.

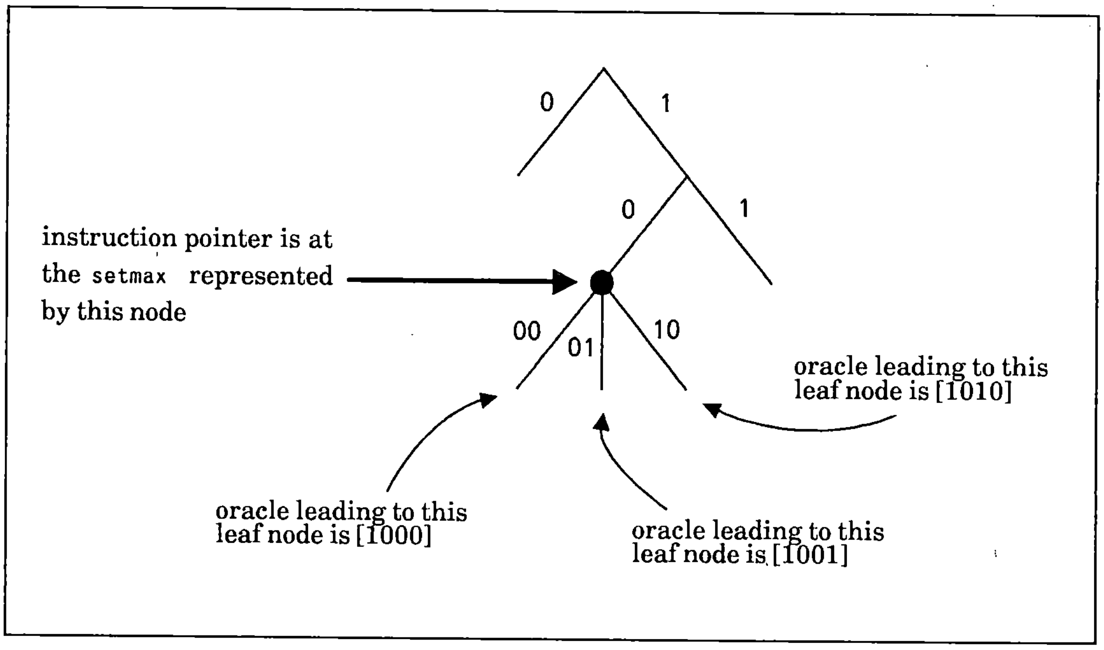
4.5 Setmax Instruction

One final instruction needed to implement any of the Delphi control strategies is setmax. Instruction setmax occurs at the beginning of each set of clauses, and has as its argument the number of clauses contained in that set. Even for sets with only one clause in them, the setmax instruction is still necessary. The setmax informs the Path Processor that a choice point has been reached. The onum instructions following the setmax can be interpreted in a variety of ways depending on what the current control strategy is. For example, assume that a non-backtracking control strategy is being executed where each Path Processor can only follow an oracle that it has been given. In this strategy, setmax marks the position in the instructions where the given oracle is decoded to find out which clause should be chosen next. Consider the tree in Figure 4.1 and the oracles which describe the three paths from the root of this tree through to each of the leaves.

Assume that a Path Processor is following the oracle [1010], and has arrived at the node indicated in Figure 4.1. The intermediate instructions representing this situation are shown:

```
setmax       3     INSTRUCTION POINTER IS AT THIS SETMAX
onumsing     3     1     entry point of clause number 1
onumsing     3     2     entry point of clause number 2
onumsing     3     3     entry point of clause number 3
setmax       1
onumsing     3     1     entry point of clause number 1
```

The next clause number to be chosen by the Path Processor can be decoded only with the knowledge that there are three possible clauses in the set to choose from. This information is given by the setmax instruction. The Path Processor then knows that the next two bits of the oracle string



0
1
0
1
instruction pointer is at the setmax represented by this node
00
01
10
oracle leading to this leaf node is [1010]
oracle leading to this leaf node is [1000]
oracle leading to this leaf node is [1001]

Figure 4.1 Using the `setmax` Instruction

must be used in determining the next clause to choose. The Path Processor is following the oracle [1010] and has already arrived at the position specified by the oracle [10]. The next two bits (10) are used in determining the proper clause to choose. In this case, it is the right-most of the three possible branches. The instruction pointer is then updated to pick the third clause in the set by offsetting to the required `onum` instruction. The argument at the offset indicated by:

((clause number picked off the oracle - 1) $\times$ instruction length of an `onum` instruction)

will then lead to the entry point of the desired clause.

4.6 Sets of Clauses

Instruction `jumponspecial` was the name given to the entry point instruction for every predicate (excluding those with an arity of zero). Special refers to the type of the first argument that is to be matched. We can consider the register to be matched with as the input argument R. If the contents of R is anything other than a variable, then it is considered a special input argument. If R is a variable, then it will be considered a general input argument. When a special argument needs to be unified with candidate clauses in a predicate, one of the pointers of the `jumponspecial` will be followed. If the input R was a general input, then no branch is taken, and the instruction pointer falls through to the next instruction after the `jumponspecial`. This is the beginning of the set of all clauses or *plenary set*. All of the other sets contained within the indexing instructions are subsets of the plenary set. In a zero arity predicate, no `jumponspecial` instruction is used, and the entry point to the predicate is the plenary set (and only set) for that predicate. There are eight different types of set for predicates which have an arity greater than zero. These sets are listed in Table 4.1.

Set	Contents
plenary	all of the clauses of a predicate
nil	clauses in the predicate with an empty list as their first argument
list	clauses in the predicate with a non-empty list as their first argument
constant	clauses in the predicate with a constant (which is not the empty list constant) as their first argument
structure	clauses in the predicate with a structure as their first argument
integer	clauses in the predicate which contain an integer as their first argument
default	clauses in the predicate which contain a variable as their first argument
fail	when there are no clauses in a set, this is the code which implements a fail

Table 4.1 Indexing Sets

The plenary set contains all of the clauses in the same order as they occur in the predicate. The five special sets are the sets pointed to by the `jumponspecial` arguments. The default set is called this since any special input will match the first argument of each of these clauses. This is because each clause in the default set has a variable as its first argument. The fail set is not really a set at all, it is just the location of the code which executes a fail instruction. Appendix 4a contains examples of most of the indexing sets.

For the following discussion we assume that a backtracking control strategy is being used. The format of the plenary, nil, list and default sets is similar to the example intermediate code shown in

Section 4.5. This format is called the standard format for a set. The first instruction in a standard format set is `setmax`, and all of the following instructions are `onumtry`, `onumretry`'s, and an `onumtrust`. If there is only one clause in the set, then a `setmax` with an `onumsing` instruction directly following it is the standard format. The other three sets (constant, structure and integer), all contain a hash table which further subdivides the clauses with that type of argument. Each of these *hashed sets* starts with a `switchon` instruction which describes the type of argument that is to be hashed. The format of these instruction is:

```
switchon<type>  arg1    arg2
```

Where type is either `constant`, `structure`, or `integer`.

The first argument of a `switchon` instruction is the address of the hash table used to subdivide the clauses. The second argument, `arg2`, specifies the number of entries in the hash table. The values to be hashed are generated by the compiler as shown in Section 4.2. The hashing function and an example of its use is shown in Appendix 4a.

4.7 Oracle Stack

In additional to the usual memory allocated for program instructions and control stacks (such as the heap and the trail), space is also reserved for the creation and maintenance of oracles. This section of memory is called the oracle stack. Figure 4.2 shows the major memory divisions of the Delphi run time Prolog system.

Program space is the portion of memory which holds the loaded intermediate code. This code includes both the compiled Prolog program to be executed, and that part of the Prolog run time system which is written in Prolog. The majority of space is allocated to a combination of the heap or global stack, and the local stack. Contents of the heap include dynamically constructed lists and structures, while the local stack holds activation records for active Prolog clauses including information needed to create choice points. The trail stack contains pointers which allow a reversible means of binding variables to data structures. When a pattern matching failure occurs, the bindings can be efficiently undone before the next clause or predicate is called.

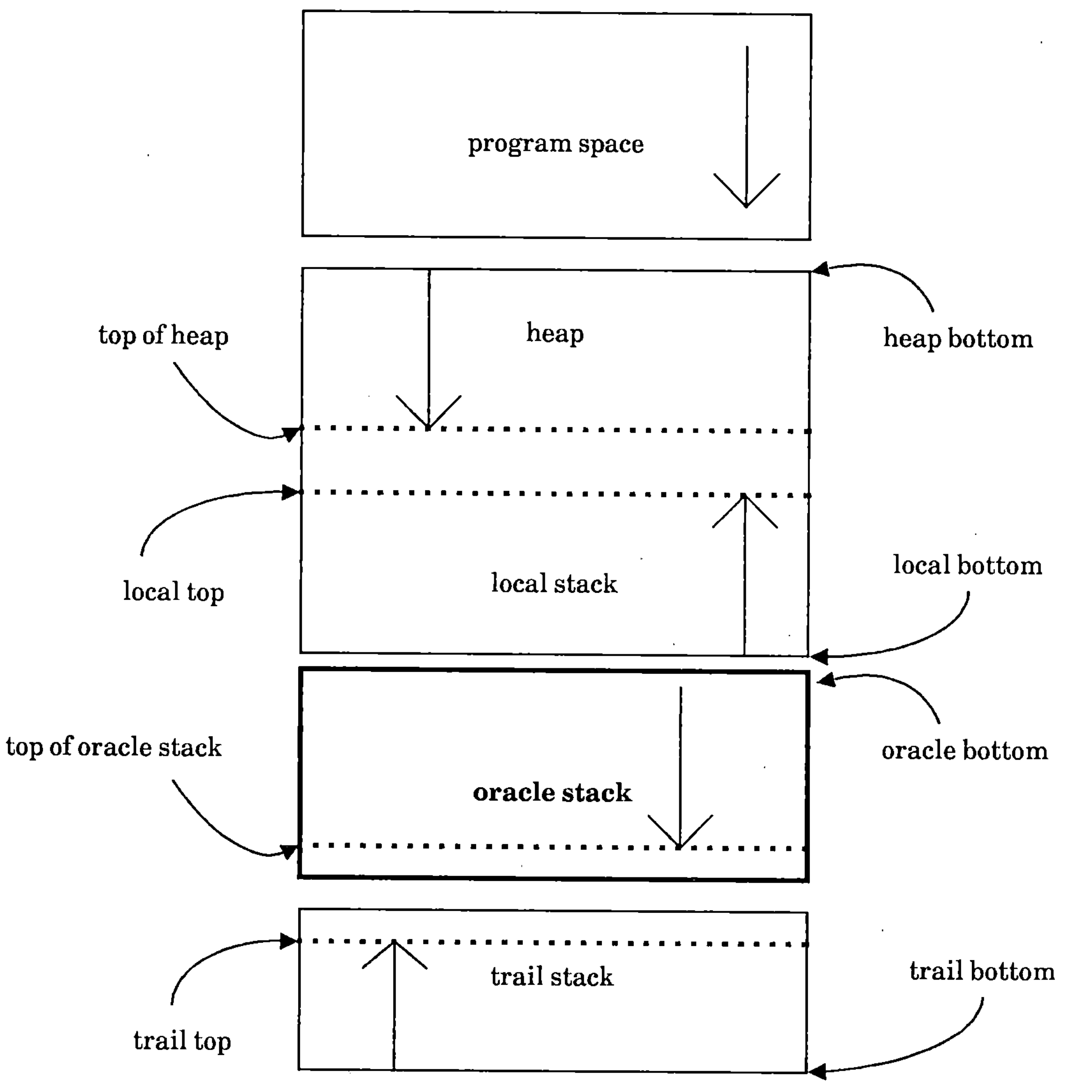


Figure 4.2 Memory Allocation

Choice points consist of special-purpose registers containing pointers to locations within the other stacks. These pointers allow the state of the machine to quickly return to a previous node in the search space when backtracking occurs. Choice points are created when there exists more than one choice or clause within the set which must be explored. If pattern matching with clause number n of the predicate fails, then the state of the machine is returned to its original form (before clause n was tried), and clause number n + 1 is attempted next. When the final clause of a predicate is attempted the choice point ceases to exist.

Choice points contain pointers to the following structures:

- Arguments of the original procedure.
- The environment of the calling procedure.
- Previous choice point.
- A program space address if the procedure succeeds.
- A program space address if backtracking is necessary.
- Pointers to the top of the heap and trail before the procedure was invoked.

In addition to these familiar special-purpose pointers, a Delphi Prolog choice point also has an ORC register. ORC points to the top of the oracle stack before the procedure was invoked. The oracle stack holds any information concerning oracles which must be saved throughout calls to other procedures. When backtracking returns control to a procedure which placed information on the oracle stack, that information is available for updating. The particular control strategy dictates the use of the oracle stack. For the non-backtracking strategies, the oracle stack is only used to hold the route through the search space that the Path Processor has taken. For some of the backtracking control strategies, the oracle stack holds a structure which maintains the *current path* and implements *limited choice points*.

Chapter 5 Models of Communication

In considering any implementation involving concurrent execution, a major topic is how to communicate among the concurrent components. For a set of processes communicating across a network, message-passing systems are widely used for Interprocess Communications (IPC). Message passing provides both a means of transferring data among different host machines, and a method for process synchronisation within the system. Two message-passing IPC facilities have been used in different implementations of the Delphi machine. The first is the Amoeba-transactions-under-UNIX model of communications [Mullender 1987]. The communications abstraction in Amoeba is the *transaction* between a client and server process. The ULTRIX implementation of 4.2BSD IPC was the second IPC facility used [ULTRIX-32 Supplementary Documents: Volume III System Managers 1984]. *Socket* connections are the communications abstraction in this IPC model.

5.1 Client-Server Paradigm

The client-server paradigm is a method of interaction among processes. These processes can be running on the same host machine, or could be operating over a network within a distributed system. The IPC mechanisms described in this chapter use this client-server model of communications. In this model, an active process called the *client*, makes a request to the passive process called the *server*, which then responds to the request. The client initiates communications with the server by sending a message to it. The server, whose function it is to reply to such requests, responds by performing an action and possibly sending a message in return back to the client process. Servers are processes which offer services across a network to a variety of executing processes which will request these services. One implementation of a server is as a background process which is *blocked* waiting for a client process to make a request. The server process is blocking on an input appearing on its communications port. The process is essentially idle and can do no useful work until a message arrives on that port. Only then is the process awakened and able to do useful work such as performing a calculation and replying to the client process. A common usage of the client-server paradigm can be demonstrated with the time-of-day server shown in Figure 5.1.

A process called the time-of-day server is shown as a background process polling its communications link named port 2034. This process can do nothing else except wait for an incoming message on the port. Another process comes into existence (the client), makes a connection to port 2034 (which he knows the time-of-day server is waiting on), and sends a message requesting the date and time. The time-of-day server responds by determining the date and time and sending the results back to the client in a human readable format. The server process then reverts to the same state it was in before the request was sent, blocking until another request arrives on its port.

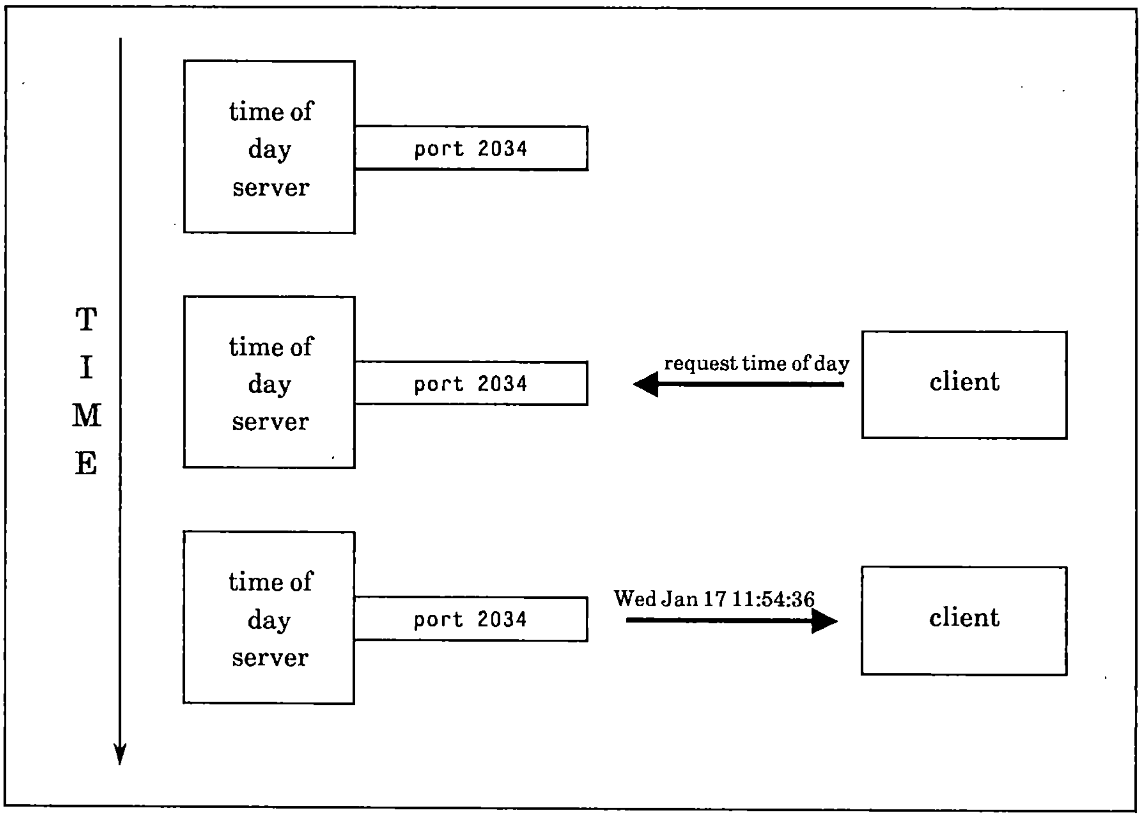


Figure 5.1 Client-Server Paradigm

5.2 Blocking and Non-blocking I/O

Blocking on input is one of the methods used by message-passing IPC mechanisms to synchronise communicating processes. The server process is halted indefinitely until a message arrives. A second process sends a request to this server and the processes synchronise. Only after this synchronisation takes place can the server process continue execution on its own. Data is written by the client process and read by the server process when this synchronisation occurs. This data is the message which is passed between the client and server processes. Message-passing IPC mechanisms can serve as both a synchronisation mechanism and a means of sending data around a network of processes executing on multiple machines.

Options may exist for the programmer to specify whether he wants blocking or non-blocking input/output. As was already shown, having the server block on an input (or read) is a very simple way to allow clients and servers to synchronise and communicate. If the server only has one task that it performs and only a single client requests this service at any time, then there is no need for providing a non-blocking read. The server has nothing else to do but answer requests from clients, so it may as well block waiting for a message to arrive. In many other applications, the tasks may not be as simple as the function of the time-of-day server. Non-blocking input or output strategies may be desirable or even necessary for the application. Figure 5.2 shows four strategies for blocking on a read or a write.

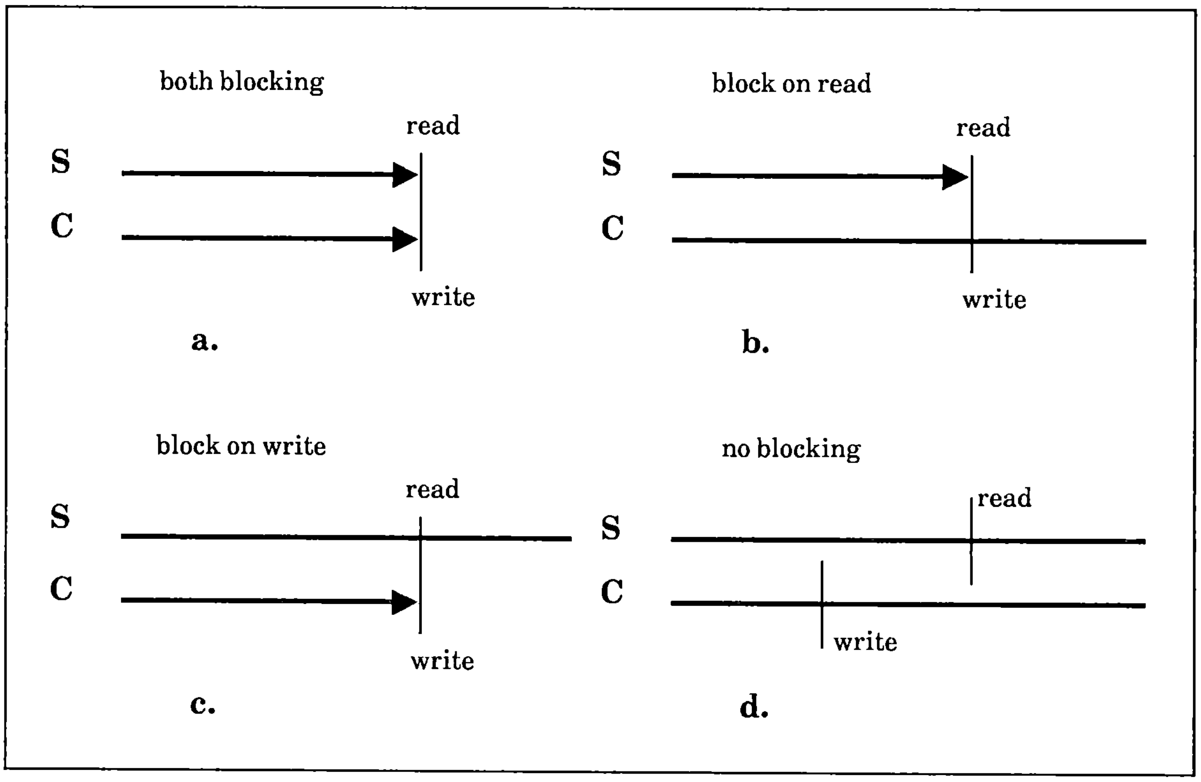


Figure 5.2 Blocking Reads and Writes

When a write occurs in some user-level output commands, the process transmits data to a buffer cache and control immediately returns to the program. This is called an asynchronous write. Figures 5.2b and 5.2d both show examples of the client process (C) performing asynchronous or buffered writes. The reads and writes occur where the vertical line touches either the server or client process lines respectively. Two reasons for providing asynchronous writes are that the client may have multiple servers which it is sending messages to, or the client may have other calculations to perform immediately after the write takes place.

The left-hand side of Figure 5.2 shows the synchronous or unbuffered writes where a client blocks until the message is sent to the device at the receiving side. This is a very clean method of the system sending messages as less or no buffer space is necessary to hold the information at each end. In Figure 5.2a both the server and the client processes block on their read and write respectively. With this input/output strategy no extra buffer space is required to hold the data which is transmitted or received. Figures 5.2b and 5.2c require buffer space to be allocated on the non-blocking side of the communications channel, the write and read end respectively. When neither reads nor writes block (Figure 5.2d), buffer space is necessary on both the client and the server sides of the communication channel.

One common implementation which will permit any of the I/O strategies described is to have buffers for both of the communicating processes. This is a general format and an efficient method of providing communications among processes running on time-sharing machines. Figure 5.3 shows the activities that take place when a read or write is performed with buffering on both sides of the communications channel. On time-sharing machines this means that other processes can continue to run until the information is available on the receiving end of the transmission. It also allows the writing side to immediately return as soon as the information has been copied to a system space buffer. Figure 5.3 demonstrates an implementation of a system which copies a buffer from the user address space to the system address space before the data is sent to the server process. It is not necessary in general to copy a system space buffer to avoid blocking other processes on the machine; this example is an implementation detail of some systems such as the BSD version of UNIX. The read and write commands are taken from the ULTRIX.documentation [ULTRIX-32 Programmer's Manual: Sections 2,3,4, and 5].

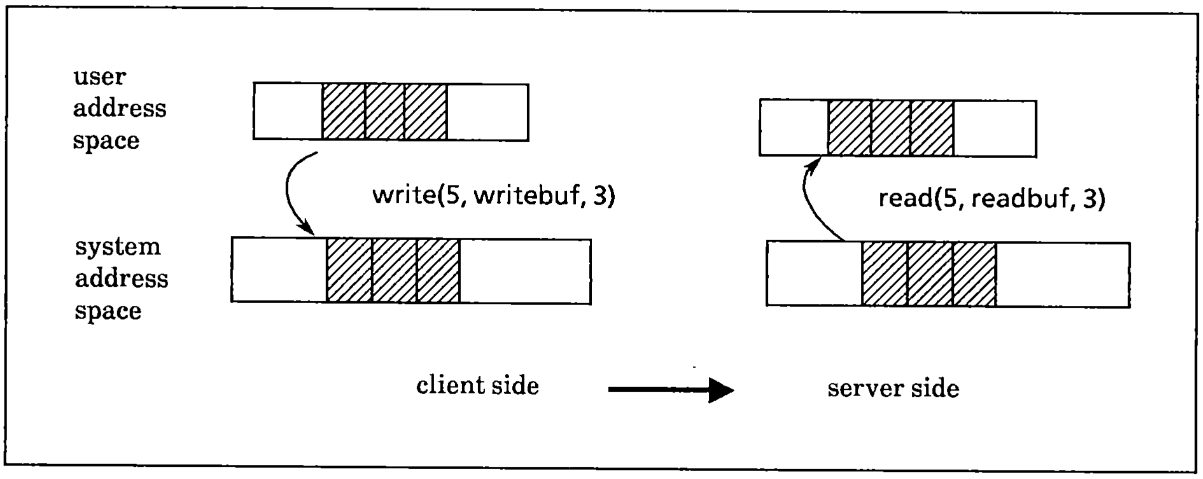


Figure 5.3 Buffering on Both Sides

Inefficiencies caused by this type of buffering are the copying from the user area into the system area on the write side, the context switches that take place, and the copying of the system buffer back to the user area on the server or receiving side. An example of a system which does not require this buffering is a process written in the programming language OCCAM running on a network of transputers [Burns 1988]. One use of synchronous reads and writes could be for real time processing on a non-time-sharing machine. Here a call to a read or write causes a context switch. The process which is still performing the computations is always the running process while the other process is blocked waiting to send or receive data. Extra care has to be taken to avoid deadlock when using synchronous I/O.

Pipes are an example of combining asynchronous reads and asynchronous writes. Using the client-server model, a pipe begins by the client performing asynchronous writes to a port (this is the buffer, or pipe), while the server is blocked on a read from that same port. When the pipe is full, the writer (client process) is blocked and asynchronous reads occur by the server until the pipe is emptied. When the pipe is empty the server blocks on a read. This process continues until an end of file condition occurs.

Asynchronous reads and writes may allow clients and servers to check the status of multiple ports without blocking. Checking the status of multiple ports could be beneficial when implementing a client which has connections to multiple servers and often writes to each of these servers. Each of the buffers which hold information from an individual port can be checked to see if a write could be performed on that port without the client blocking. A free port could then be selected to write to, and the client would send information to that server. An instruction which allows the monitoring of multiple ports is the `select` call provided within ULTRIX. The ULTRIX implementation of 4.2BSD IPC allows any of the I/O strategies which have been described; the Amoeba IPC facilities do not.

5.3 Amoeba IPC

The Amoeba Distributed Operating System [Mullender 1987] was the first system which was investigated for running the Delphi machine. Amoeba offered a simple user interface coupled with potentially fast communications. The first decision to be made was between using the native Amoeba system and the Amoeba-transactions-under-UNIX implementation. This was a very simple decision as the very basic services that were needed for a full Delphi machine were not yet available with the native Amoeba implementation. These facilities included the ability to read and write files (a file server), a clock for timing results, a remote file distribution service, and many standard UNIX library calls. Amoeba-transactions-under-UNIX uses the IPC mechanism of Amoeba but runs under the UNIX operating system. This means that all of the facilities available on a UNIX system would be available with this implementation.

The alluring feature of Amoeba is the simplicity in the user-level code needed to send messages across a network. Figure 5.4 gives an example of a simple successor server written using Amoeba-transactions-under-UNIX. When sent an integer, this server replies by sending back the successor to that integer.

The code demonstrated in Figure 5.4 is an example written to succinctly demonstrate a simple server. Many deficiencies such as the lack of error checking, and having a port *hard-wired* in the header, would not be used in a real server. With that caveat, this example is a real runnable server which can be started on any of the available host machines and left to run for as long as the machine remains up. A client could at any time request this service by sending an integer I to the port named p1, and the server would reply by returning the value of I+1. A client program which uses this server is shown in Figure 5.5.

If the successor server program is executing on one of the available host machines (running Amoeba-transactions-under-UNIX), and this client program is started on the same or any other host machine, the results will be to print out "i = 8".

```
#include "/usr/amoeba/h/amoeba.h"

main()
{
    header hdr;
    int i;

    strncpy(&hdr.h_port, "p1", PORTSIZE);

    while (TRUE){
        getreq(&hdr, &i, sizeof(i));
        i++;
        putrep(&hdr, &i, sizeof(i));
    }
} /* end main of successor server */
```

assigning a port to send to

body of the server which is performed forever

Figure 5.4 Amoeba Successor Server

```
#include "/usr/amoeba/h/amoeba.h"

main()
{
    header hdr;
    int i = 7;

    strncpy(&hdr.h_port, "p1", PORTSIZE);

    trans(&hdr, &i, sizeof(i), &hdr, &i, sizeof(i));
    printf("\ni = %d\n");
} /* end main of successor client */
```

assigning a port to receive from

initiating communications

Figure 5.5 Amoeba Successor Client

Ports are the communications addresses which connect the server and client processes during a transaction. In the version of Amoeba we used, the port is a six-byte number chosen by the server itself and placed in its proper position in the header structure. In the example client and successor server, this was the only field which was filled in the Amoeba header. Most of the fields in an Amoeba header are used to manipulate objects and to access the privileges that a user may have for a particular object. The Amoeba header structure is shown below:

```
typedef struct {
        port    h_port;                       /*server port*/
        port    h_signature;                  /*authentication*/
        private h_priv;                       /*private part, for object manipulation*/
        unshort h_command;                    /*operation code*/
        long    h_offset;                     /*offset in the object*/
        unshort h_size;                       /*size of buffer*/
        unshort h_extra;                      /*extra parameter*/
} header;
```

This header is one of the two components of an Amoeba message. A transaction consists of the client sending a message to the server and the server sending a message back to the client. These messages both have a header part and a user buffer part. This user specified buffer is not mandatory so sometimes a null buffer is sent in a message. Often enough information can be placed in just the header portion of a message so the buffer need not be used.

The first field of an Amoeba header contains the port of a server process to which this message is being sent (for a client process). With a server process the port field contains the name of the port on which the server waits for incoming requests. The method by which these ports are assigned demonstrates one of the user friendly interfaces to the Amoeba IPC. Amoeba allows a user to dynamically name ports and initiate servers. Before this facility is described, a brief introduction to the Amoeba model is given.

Amoeba is a capability-based operating system which provides its users with a means of manipulating processes, files and any other abstract data types by using *capabilities*. The data types which are manipulated are termed *objects*, and these objects have capabilities associated with them. Capabilities function as a protection measure (by being sparse and encrypted) to ensure that users do not interfere with each others objects accidentally or maliciously.

Servers are used as the middlemen between a client process and the objects it wishes to have access to. A server may have control over one or many different objects; this depends upon the particular application. In the Delphi implementation capabilities were used to specify and protect particular processes. Many of the protection mechanisms available within Amoeba were not needed in the Delphi implementation. The major reason for this is that Delphi was the only project at the Computer Laboratory which was using the Amoeba system. The capabilities were still needed, but only to protect Delphi processes from other Delphi processes. The capability data structure is shown below. The port field is a six-byte character field:

```
typedef struct {
        port    cap_port;
        private cap_priv;
} capability;

typedef struct {        /* private part of capability */
        char    prv_object[3];
        char    prv_rights;
        port    prv_random;
} private;
```

Capabilities are divided into two parts. The first is the server port which provides information on where a call should be placed to access the object which is desired. The second part contains all of the protection and identification fields to manage the use of this object. The first private field, prv_object[3], can be used by the server to identify this object from any of the others that it manages. The prv_rights field is used as a mask for identifying the operations which the user is allowed on the object; such as read or write on a file object. The prv_random field can be used as a protection mechanism by the server to ensure that the capability which has been shown to it is valid. It is up to the designer of the server just how the access and protection of the objects should be implemented; Amoeba provides the framework.

Names and capabilities for objects appear to the user in a *global* directory space accessible from any of the host machines. This directory is provided by a directory server running on one of the host machines. When an object is appended to this global directory, its name and capabilities are incorporated into this structure which is similar to the UNIX directory facility. The object names are placed in the directory of the creator within this global directory structure. The creator's directory is a subdirectory of user, which is a subdirectory of root. An example of this directory structure containing object names and capabilities is shown in Figure 5.6.

This directory structure is examined by using the command list. When used with no parameters this command lists all of the objects in the user's home directory. The listing contains the name of the object which is a text string, and the capabilities which that user has for accessing the object. The first field in the capability portion is the port of the server which is responsible for manipulating the object concerned. The other fields are protection and rights of access for the object.

using command `list` in directory `csk`

```
name of object      port of server      private part
list
root                dirsvr              2 ff ???U??
pool                dirsvr              5 ff ????19
DelPhiNET           dirsvr             85 ff ?\????
echo                ???C??              0  0 ???>o?
list DelPhiNET
OracleSv            u?????              0  0 ??????
AnExSv              =?????              0  0 ?R+#8?
P3                  Ku??k{              0  0 m?)?y?
P1                  #0J???              0  0 "??k8?
P2                  c????/              0  0 ?<&???
P5                  va%??G              0  0 ?t????
P6                  ?Sv???              0  0 W???k?
P7                  j????f              0  0 ???T@?
P4                  ,??E??              0  0 ??X???
P8                  '????R              0  0 j#????
P9                  ?6]7??              0  0 h?s???
P10                 ?-#b??              0  0 ??$)??
P11                 ??)??K              0  0 ?(0A??
```

```
list root
user      dirsvr     3 ff G?c??H
pool      dirsvr     5 ff ????19
public    dirsvr    22 ff ?'????
list root/user
mb        dirsvr     1 ff ??????
csk       dirsvr     7 ff i?$;??
gem       dirsvr    10 ff ??:?q?
```

```
list root/user/csk
root        dirsvr     2 ff ???U??
pool        dirsvr     5 ff ????19
DelPhiNET   dirsvr    85 ff ?\????
echo        ???C??     0  0 ???>o?
shout       w?Km??     0  0 ?3??1i
```

Figure 5.6 Global Directory Structure

As shown in Figure 5.6, the server's port field is either the word `dirsvr` or the ASCII string resulting from encoding the port name. A port field with `dirsvr` indicates that this object is itself a directory server containing all of the objects that appear below it in the hierarchical structure. To list all of the objects located below a directory object, the list command is given with a parameter specifying the pathname for that directory. By altering the access rights to directory structures and other objects, a complete protection system can be enforced upon the users of those objects.

5.4 Transaction Primitives

A *transaction* is a complete circuit of communication consisting of: a request to a server, the receipt of this request, a reply from the server, and the receipt of this reply. Amoeba provides three primitives for use in a transaction between a client and a server process: `trans`, `getreq` and `putrep`. A `trans` or transaction is the initiating communication by the client process. The `getreq` (get request), and `putrep` (put reply) primitives are used by the server. The syntax for these three is given as:

```
unsigned short    trans(header1, buffer1, size1, header2, buffer2, size2)
header            *header1, header2;
char              *buffer1, buffer2;
unsigned short    size1, size2;

unsigned short    getreq(header, buffer, size)
header            *header;
char              *buffer;
unsigned short    size;

unsigned short    putrep(header, buffer, size)
header            *header;
char              *buffer;
unsigned short    size;
```

A server process blocks on a `getreq` waiting for a message to arrive on the port specified in the header structure. When a message arrives the header and buffer structures will contain the header and buffer of the sending process' transaction. Since the ports must have been identical in both header structures (of the client and server), the port field remains the same throughout the transaction. If no client process attempts to contact the server on its specified port, the server process remains blocked indefinitely. The Amoeba IPC does not provide a non-blocking `getreq` command. When a message is received, control returns from the `getreq`, and the server process is free to continue its computation. At the end of this computation a `putrep` is called, and a message (header and buffer) is returned to the client process. In a `getreq`, the `buffer` and `size` parameters give the location and the size of the buffer where data from the client process' buffer is to be placed. The returned value is the actual size of the buffer that was transmitted from the client, and is always less than or equal to the initial `size` value. Any additional bytes greater than `size` will be discarded.

Client processes also block when they perform their `trans` calls. A `trans` procedure attempts to send the message contained in `header1` and `buffer1` to the server whose port is specified in the header. It then remains blocked until a reply is sent by the server. This reply is placed in `header2`, and `buffer2`. In Figure 5.5, the successor client has its sending and receiving buffers at the same address. In this case it was unnecessary to retain the initial information sent (the integer 7), so it was overwritten when the reply came back. Clients, unlike servers, do not indefinitely block on the port placed in their header. If a port cannot be located within the time specified in the `timeout` call, an error is returned. Even with the `timeout` call, the transaction protocol is an example of synchronous transmission; the client and the server are synchronised with each other when the data is sent and received.

Appendix 5a shows an example client and server process which use the Amoeba IPC mechanisms. This code provides a better look at the transaction primitives of Amoeba, and the use of the capability mechanisms. The capability mechanisms are used in this example code in the same way as it is used in the Delphi implementation; for avoiding unintentional servers and clients being proliferated across the network. The timeout procedure shown in the routine `trans_C_output` is defined as follows:

```
                            timeout(deciseconds)
     unsigned short         deciseconds;
```

The `trans` in routine `trans_C_output` (see Appendix 5a) waits for a maximum of ten seconds (the time is measured in tenths of a second) for locating the port connection specified in the header. If the port is not located in that time, a timeout occurs. The error condition shown in case number two is then reported. A zero value is used if the client is to try and locate the server port for an infinite amount of time.

5.5 Dynamic Port Allocation

Earlier in this chapter it was stated that assigning a hard-wired port name for a server is not a good idea. The successor server code in Figure 5.4 is an example of using a hard-wired port placed in the header. Why this method is unsound will now be discussed along with how to correct the dangers through the use of capabilities.

Consider the standard Delphi model where we have one Controller process and any number of Prolog (server) processes on the same or on separate host machines. If we adopt the system of using hard-wired ports then the Controller must know in advance the maximal amount and names of all Prolog ports. One way to avoid needing this information is to associate the name of host machine with the port name. The Prolog on host number one has port named "host1", the Prolog on host two has port name "host2", up to the Prolog with port "hostN". With this system for naming ports the Delphi model has been restricted. The assumption made is that there is one Prolog per host machine, and this is not one of the specifications of a Delphi network. An updated port naming system could be adopted where an additional letter is added to the port name to uniquely identify a Prolog executing on the same host. If there are three Prologs on host number one, their port names would be "host1a", "host1b", and "host1c". This does define a unique naming convention for ports, but does not help the Controller find out how many Prologs exist, or which port names are being used.

Port names come into existence when a Prolog server is started on a host machine; the Prolog creates or acquires a port designator for itself on initialisation. In the simple successor server, the port name "p1" was chosen and hard-wired as this server's port. There are numerous problems with acquiring a port name in this way. Firstly, in Amoeba, no port name is special. Two servers can be listening on the same port and any number of clients can simultaneously write to a particular port. The problem this poses is one of security against malicious intruders, and protection against accidentally initiating more than one server with the same port.

Protection against malicious abuse includes disallowing other users' servers to masquerade as one of your own servers. When ports are hard-wired, it is a trivial task to read the proper user's server code, get the port name, and start a false server which listens on the same port name. For the successor server, a false server process could listen for communications on port "p1", answer back with the proper response (so the real user's of that service do not know that they are communicating with an intruder), and then perform whatever malicious deeds it was designed to do. This is one of the ways in which Trojan horses and viruses can be running on machines without anyone being aware of the fact for a long time. Two situations with false servers can occur:

(1) If the false server is started in place of the real server, then all communications intended for the real server will be sent to the false one.

(2) If the false server is started in addition to the real server, then it is unspecified as to which of the servers will get any particular message.

In the second case, it is unlikely that a user of this service would notice anything wrong as the proper answer will be supplied by either the false or the real server. However, a system manager or vigilant user might notice the existence of the two server processes unless an extra effort was made by the intruder to disguise its process status too. The second case also demonstrates a problem that does not involve any malicious intent—the accidental occurrence of duplicate servers using the same port. During the testing phase of a server's life, it is common to start the server, test it, and kill it so that alterations can be made. When using a hard-wired port, the modified server will be listening on the same port. If the original (or any other intermediate versions of the original) server is not killed, then multiple servers could be outstanding listening on the same port. It is indeterminate as to which server will receive any message sent by a client. This is particularly a problem with different versions of the same server as some of the older versions might not perform the proper tasks. Even if the server has been completely debugged, two executing copies will waste resources on the host machines. There is, however, a use for having multiple servers listening on the same port. Duplication of a server on more than one host machine is a common method of providing reliability in a distributed system. With two identical servers, one will remain up even if the other server machine is down.

In addition to the problem of unwanted multiple servers, another fundamental difficulty arises from allowing hard-wired port names; how does the Controller know what servers are available? Without a centralised service to supply the port names in the first instance, the Controller will not know what these names are. One solution is to have a file containing all of the potential host machines and use the port naming scheme as described above. The Controller could then perform a transaction using each potential port name. For every host machine name in the file, the Controller would try out the ports "hostname1a", "hostname1b", and so forth, until a transaction timed out. This procedure could continue until all of the possible port names are queried in order. The problem with this method is that it does not allow for the possibility of a new server starting up after the Controller has tried the port name that the server will listen on.

Using the capabilities mechanism provided by Amoeba prevents many of the problems inherent in using hard-wired ports. Four additional commands are provided by Amoeba to protect against duplicate port allocations and allow servers to dynamically create their own port names. The sample Amoeba programs in Appendix 5a demonstrates the use of capability commands to safely create and advertise port names. Procedures `server_put_capability` and `get_server_capability` demonstrate the capability commands provided by Amoeba. The call to `uniqport` fills the given port structure with 48 random bits. It is unlikely that this generated random number will duplicate a port name already in existence. The random number in the private part of the capability is also generated for use in authenticating a user's capability for that object. The syntax for `uniqport` is:

```
uniqport(newport)
port  *newport;
```

Three routines are used to update the global directory structure maintained by the directory server. This global directory holds the names and capabilities for all known objects. For the Delphi implementation, the objects included in this directory space are the Delphi processes (the Controller and Prolog processes). This same system is used in the example programs provided in Appendix 5a; the objects are all processes. Amoeba command `am_lookup` attempts to find an object name in the directory structure. Deletion of an object name and capability is performed by `am_delete`, and `am_append` adds a new object name and capability to the directory space. The syntax for these three commands is:

```
int             am_append(objectname, capabilityforobject)
char            *objectname;
capability      *capabilityforobject;

int             am_delete(objectname)
char            *objectname;

int             am_lookup(objectname, capabilityforobject)
char            *objectname;
capability      *capabilityforobject;
```

An example of the echo server's entry in the global directory space is shown in Figure 5.6. Also shown are all of the capabilities created for a configuration of the Delphi machine. The Delphi machine capabilities are listed under the directory DelPhiNET.

As the size and quantity of Delphi processes grew, the Amoeba system started crashing frequently. When up to one half of the code in the smaller clients or servers was dedicated to trying to catch Amoeba system errors, or avoid them, a new Delphi machine was implemented. This new implementation contained the IPC facilities of the Berkeley Software Distribution of UNIX, version 4.2 (4.2BSD).

5.6 IPC using 4.2BSD Sockets

Sockets are the communications abstraction used in the 4.2BSD (and 4.3BSD version [Leffler, McKusick, Karels and Quarterman 1989]) Interprocess Communications (IPC) facilities. Sockets provide a very general IPC mechanism which allows great flexibility in the types of systems which can be built from them. They function within two standard domains supplied by the system; with the allowance for any additional communications domains to be supported by the user. The UNIX domain allows sockets to have names similar to UNIX pathnames, but is generally used only for processes communicating on the same host machine. Pipes for example, could be implemented using UNIX domain sockets (and are in BSD). Internet is the domain used by processes which communicate across a network. Internet is itself divided into two protocols: UDP (Unreliable Datagram Protocol), and TCP (Transmission Control Protocol). All of the following discussion assumes the use of TCP as this is the underlying protocol used by the Sockets implementation of Delphi.

Sockets are the endpoints of communication channels which are set up between processes to allow them to send and receive messages. For a pair of communicating processes, each process sets up one socket, and then proceeds through a standard set of commands in attempting to connect two endpoints together and form a communication link. In setting up a communications link between processes, the client-server paradigm is used. Separate commands are required depending on whether the process is initially viewed as a passive server process, or an active client process. After the channel has been set up processes are free to send and receive communications in any specified order. Appendix 5b has a pictorial representation of the system commands needed to make a connection between a client and a server process. Creating a socket connection provides a two-way communications link between the client and server process.

Establishing a connection implies that there is a service available, and that the client knows the host machine and port number on which to find the required service. A server creates an endpoint of communication using the `socket` system command and proceeds to `bind` this socket by specifying the host name and port number at which this service can be reached. Command `listen` sets up a queue for client processes waiting for their requests to be handled by the service. The `accept` command causes the server to block on the named port waiting for incoming traffic.

A client process only has to perform two system commands before a connection to a server can be established. Command `socket` creates an endpoint of communication, and `connect` does all of the rest. For a client process to perform `connect`, it must know both the Internet address of the remote host, and the port number on which the service resides. In the 'shout' client example shown in Appendix 5a, two command-line arguments are given to the client process. These two arguments are needed to establish the server's identity and location on the network. The host machine's Internet address is returned by `gethostbyname`. The port on which the service listens is returned by the `getservbyname` system command. This port is a number which could have been hard-wired by directly putting the value into the `sp->s_port` variable. There is also a method for using a predefined name (associated with a port number). An example of starting the 'shout' client program is with the command `shout path01 test2`, where `path01` is the host machine name, and `test2` is the name of the port where the required service can be found.

TCP provides stream socket connections with reliable delivery of all messages. A stream has the property that the bytes sent at one end of the communications link will be received at the other end exactly as they were sent. Once a connection has been established, the processes are free to send messages in either direction along this bidirectional communications socket. This process of sending and receiving messages can continue until the channel has been closed. Appendix 5 has a pictorial representation of the system commands used in sending and receiving data between a client and server.

A comparison is now given between the transaction commands of Amoeba and the reads and writes using sockets. A transaction entails the entire process of a client initiating communications by sending a message to a server and having that server respond. The equivalent code for a system using sockets is to have the client initiate communications with a `write` to a socket, and the server receive this message with a `read` command. Then, the server must respond with a return message by writing to the socket, with the client now receiving this message with a `read`. This involves four system calls and therefore four context switches to execute the code. It might be expected that the Amoeba system would show a better performance than the generalised IPC using sockets. Amoeba did not show any spectacular performance, even when the kernel stayed up long enough to run an entire four queens problem.

5.7 Echo Servers

One of the original bugs in the Amoeba system was a problem with the `trans` call. The error message, "FAIL because of network or server crash" would be returned even when the network and the server were fine. The problem occurred when the following conditions were combined:

(1) There is more than one client processes competing for the same service.

(2) The clients are communicating very frequently to the server.

With the amount of communications that takes place in any of the non-backtracking strategies, these conditions always existed, and the Amoeba implementation was useless for them. While this bug was being investigated, the first backtracking strategy was implemented—*automatic partitioning only*. The automatic partitioning only strategy does not involve any control communications between the Path Processors and the Controller. With this strategy, the Amoeba system was able to run a complete Prolog problem without crashing. The frequent communications were causing many of the problems.

Some tests were performed to compare communications times using Amoeba-transactions-under-UNIX and the ULTRIX implementation of sockets. A very simple problem, an echo server, was chosen as the benchmark. An echo server receives a buffer and sends the exact same buffer back to the client process. The servers and clients to execute this problem were written using Amoeba IPC and sockets. Code for all of these programs can be seen in Appendix 5a.

An initial decision in the design of the Delphi machine was to pack the bits which represented choice points instead of having one bit per byte. This greatly reduced the size of the communications so that oracle sizes were mainly below 100 bytes (800 bits). Packet sizes from 1 byte to 100 bytes were tested with both of the systems executing simultaneously. The two servers were started on the same host machine, with the two clients started simultaneously on a different machine. The first test was to allow 100 transmissions of each of the packet sizes. The results are shown in Figure 5.7.

Though the servers and clients were started at the same time, conditions on the network could exist which would benefit one of the two IPC mechanisms. For this reason, this test with 100 repetitions was tried numerous times on different machines and at various times of the day. Though the exact data points differed with each time trial, the trend was always that the Amoeba points were slightly above the Socket data points. The Amoeba times were always more consistent than the Sockets times. Using such a small number of repetitions creates more erratic data. Additionally, the number of communications represented here is slightly lower than many of the Delphi benchmarks. The trials were performed again with the `NUMTIMES` variable in the header file set to 1000 (see Appendix 5a). These results are shown in Figure 5.8.

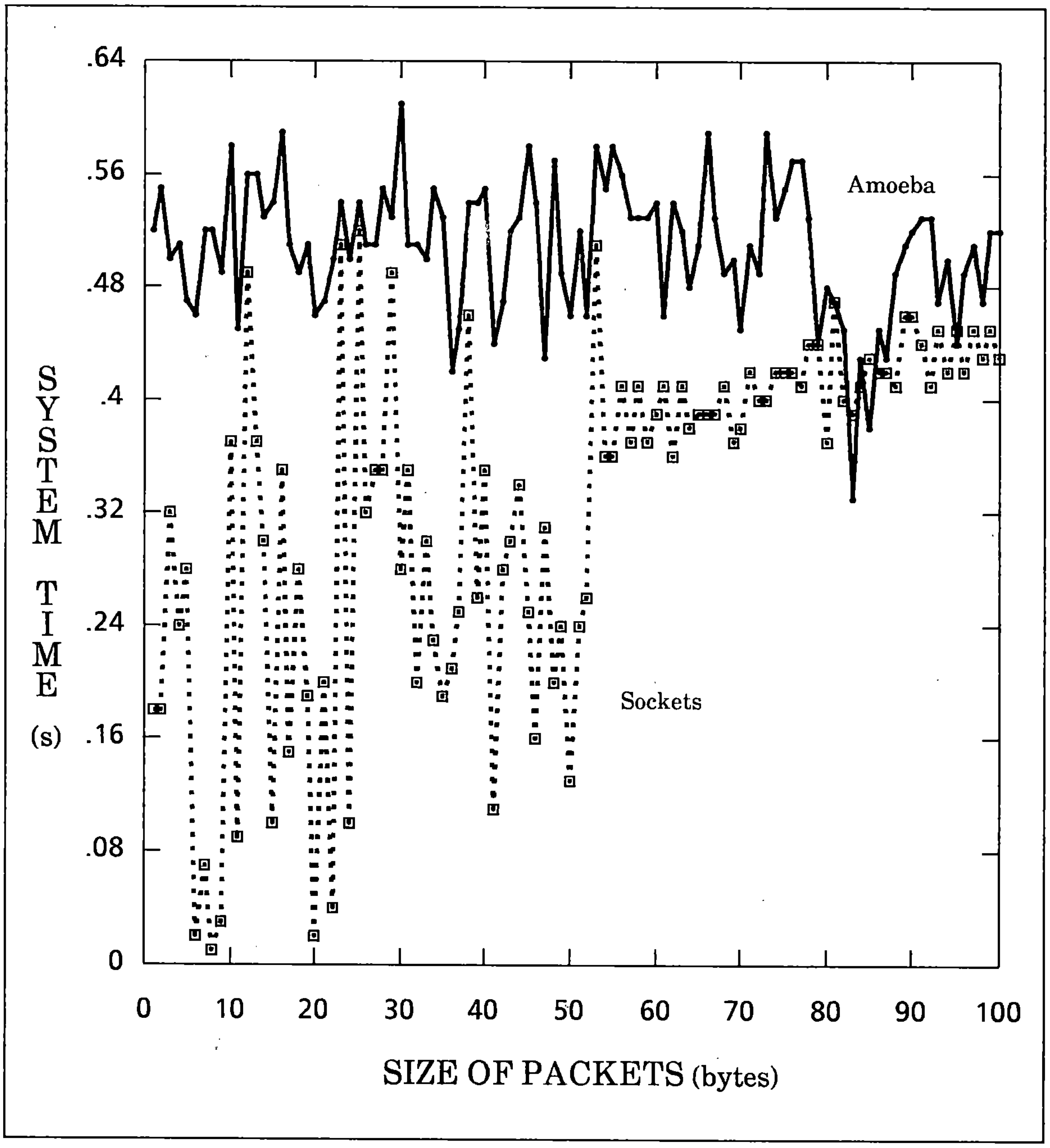


Figure 5.7 100 Packets Transmitted

During the simultaneous running of the two echo server systems, it was noticed that the Amoeba client process was taking a much longer time to complete than the client using socket connections. The communications times (shown in Figure 5.7 and Figure 5.8) were not causing the problem since they are fairly similar.

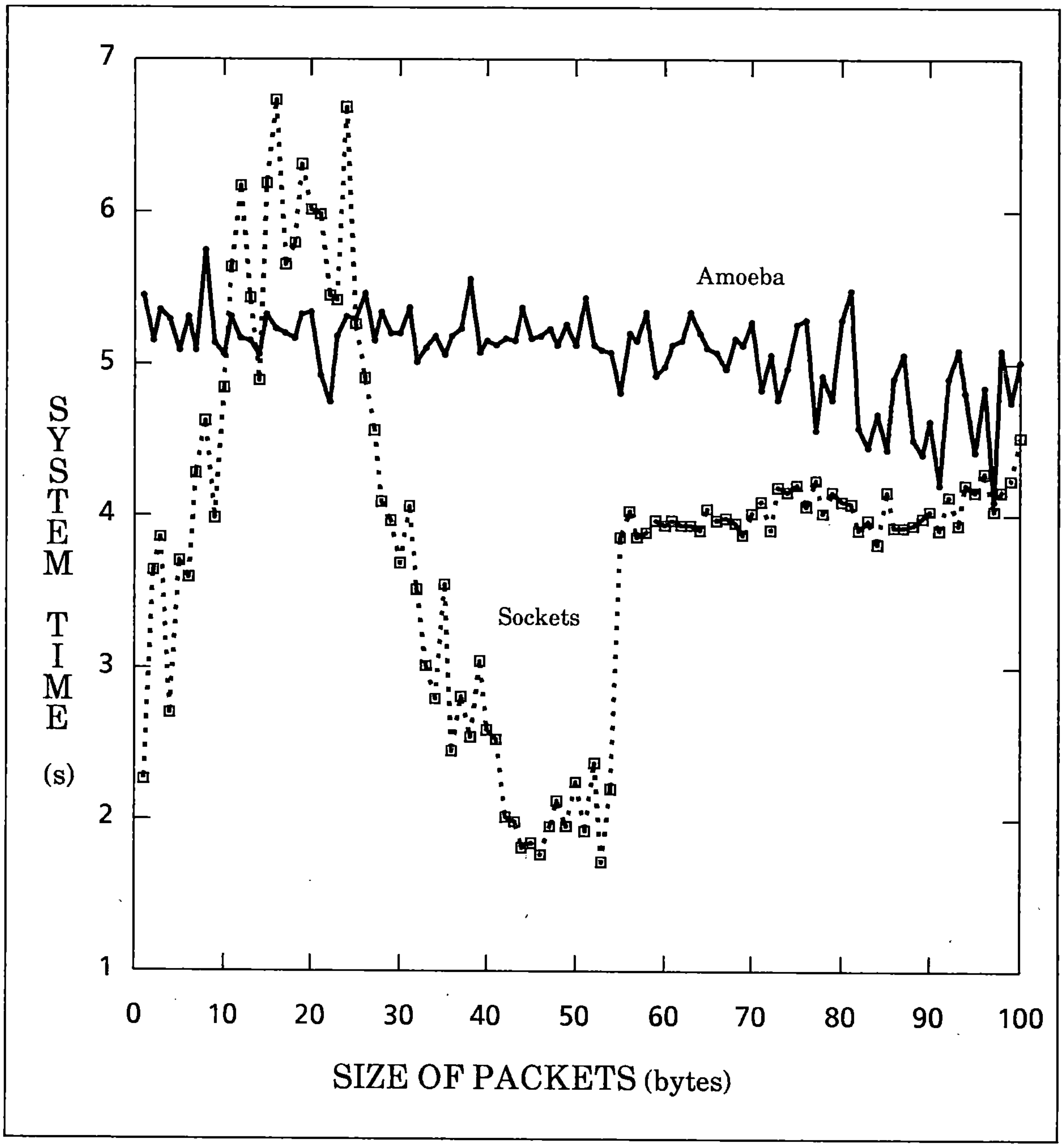


Figure 5.8 1000 Packets Transmitted

One possible explanation of this is a problem in Amoeba with the frequency of packets. The shout client process is hammering away at the echo server sending as many packets as it possibly can across the network. Perhaps the server process cannot keep up with the demand, and starts dropping the packets. This would lead to an increase in the execution times since these dropped packets have to be retransmitted by the client (Amoeba IPC ensures reliable delivery).

It is simple matter to test this hypothesis by putting a delay in the client loop. This delay would slow down the frequency of packets sent to the server. Instead of a tight loop consisting of just the trans call, an additional loop with a delay variable was added. In Appendix 5a the delay variable is called DELAY and can be seen in both the Sockets code and the Amoeba code. For Figures 5.7 and 5.8, this DELAY variable was set to zero.

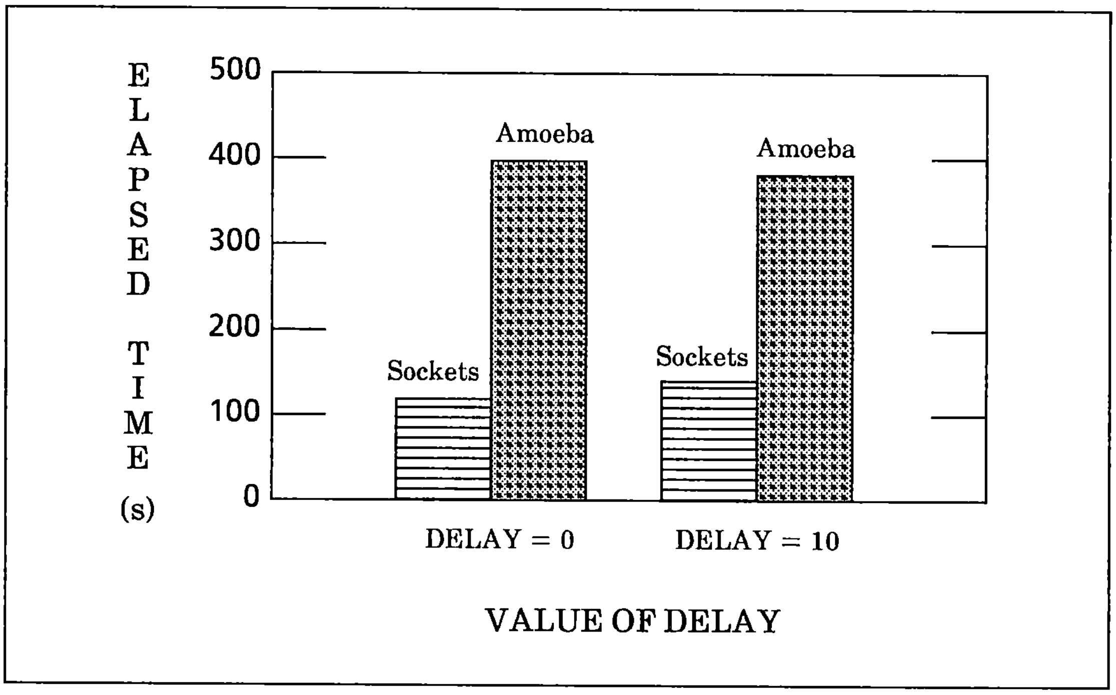


Figure 5.9 Execution Times for 100 Packets

Figure 5.9 is a bar graph showing the execution times of the Sockets and Amoeba systems. For each size of packet from one byte to one hundred bytes, both systems sent and received one hundred packets of each size. On the left-hand side are the times with no delay (DELAY set to 0) in the client loop. On the right, DELAY has been given the value 1000. The value 1000 means that the delay loop was executed 1000 times. It appears counterintuitive that when a delay is added to the client's transmission loop, the execution time decreases; but that is exactly what happened with the Amoeba system.

Figure 5.10 shows a similar plot with the number of transmissions extended to 1000 times per packet. The final plot shows the execution times for transmitting packet lengths of from 1 to 100 bytes 10,000 times each (Figure 5.11). The code for the Amoeba Ethernet driver was checked after these results had been demonstrated. A bug was discovered in the driver which caused the server process to frequently drop packets and force the client to retransmit.

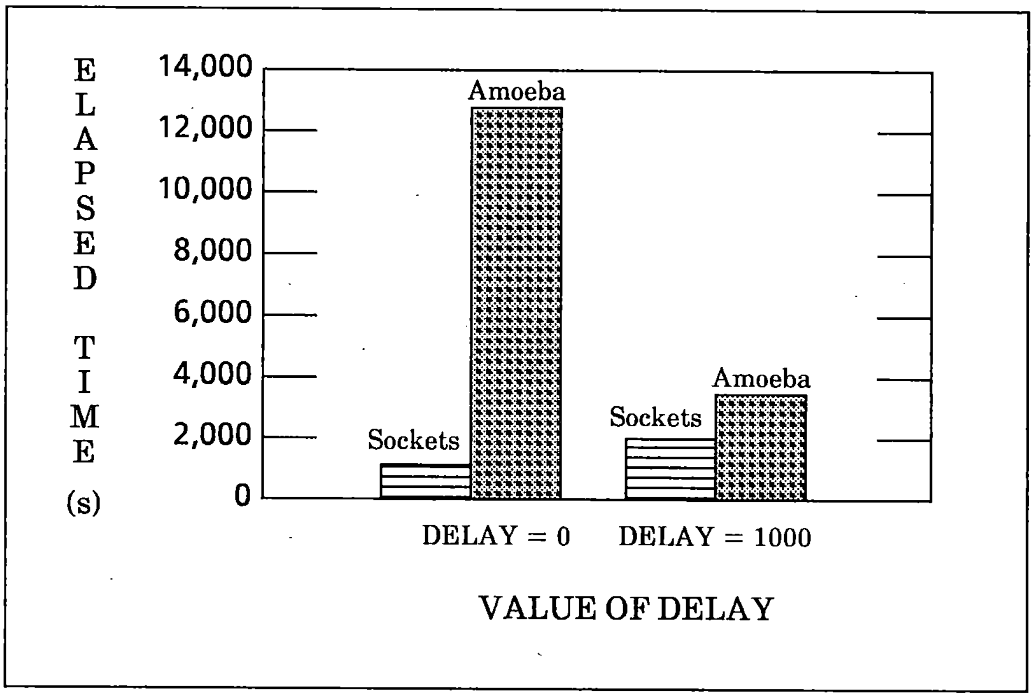


Figure 5.10 Execution Times for 1,000 Packets

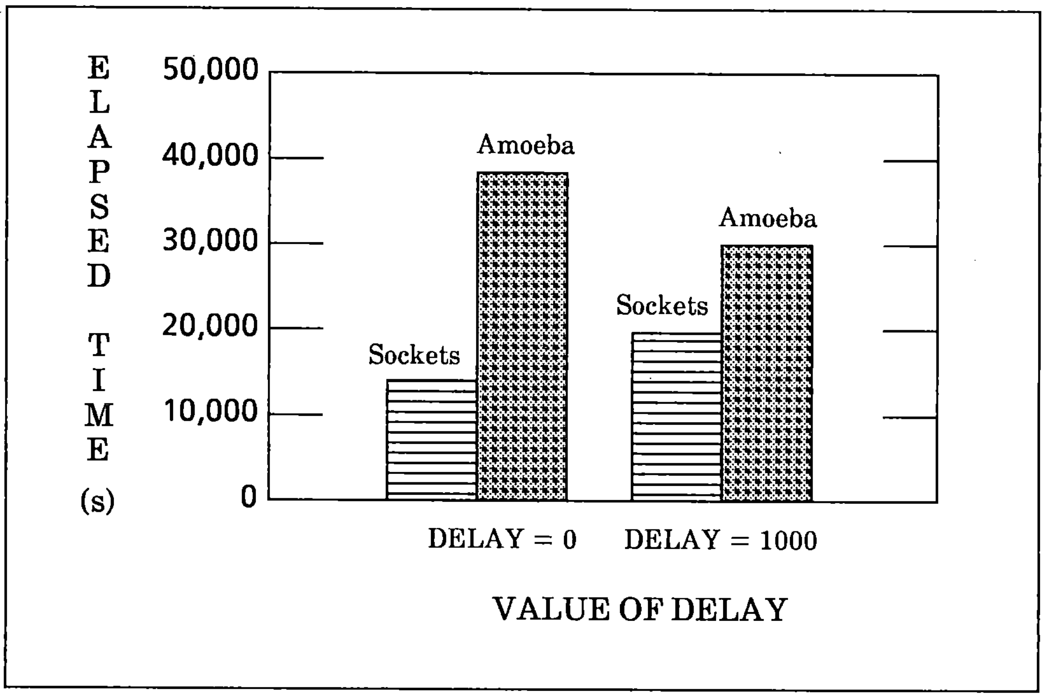


Figure 5.11 Execution Times for 10,000 Packets

Chapter 6 Distributed Delphi

Models and implementations of two distributed Delphi machines are described in this chapter. The first uses Amoeba-transactions-under-UNIX Interprocess Communications (IPC) facilities and was designed to run under the Amoeba Distributed Operating System [Mullender 1987]. The second implementation uses the 4.2BSD IPC mechanisms implemented within ULTRIX [ULTRIX-32 Supplementary Documents: Volume III System Managers 1984]. The Amoeba version of Delphi is a distributed system with multiple client and server processes operating across a network. Delphi running with the ULTRIX 4.2BSD IPC is a more integrated system with the client processes being combined into a single executable image. The process management facilities in the ULTRIX version are also incorporated in the single client. In the Amoeba version the process management system is an external system requiring additional client and server processes.

6.1 Amoeba Distributed Delphi Model

The original implementation of the Delphi machine is a model where the Controller process is divided into three component processes. In this model, not only are the Prolog processes distributed among the various machinery, the Controller itself can be distributed among any of the available host machines. The distributed Controller is divided into the following three unique processes:

(1) Oracle Generation process
(2) Jobs Distribution process
(3) Answers and Exceptions server

These processes were originally designed to handle non-backtracking control strategies. Their functionality is first described in terms of control strategies which do not allow backtracking. The Oracle Generation process or Oracle Generator is the algorithm for describing paths through the search space which need to be explored. Depending on the strategy, oracles can be both created and stored by the Oracle Generator process. In the *naïve depth-first iterative-deepening* implementation, the Oracle Generator creates and stores a single oracle at a time. When a Prolog process (also called a Path Processor or PP) becomes idle, it is given the single stored oracle to explore. The Oracle Generator then creates the next oracle in succession stores this oracle and waits for another Path Processor to become idle. With other non-backtracking strategies, a jobs queue needs to be maintained for storage of the numerous oracles which have not yet been explored.

There are two major non-backtracking strategies which have been implemented: a *bit by bit strategy*, and a *branch by branch strategy*. In the bit by bit strategy, the Oracle Generator creates new oracles when a Prolog process reaches a choice point and becomes idle. The newly created oracles are one bit extensions of the path previously explored by the idle PP. These new oracles are then placed on a *jobs queue* until they are requested. When a Path Processor becomes idle the first job on the queue is assigned to the idle PP by adding a host identifier onto the job structure residing in the jobs queue. This identifier is used by the Jobs Distribution process to ensure the delivery of the path to the proper Path Processor. The entire job structure is placed onto the *distribution list* which is picked up by the Jobs Distribution process when a transaction between these two processes occurs.

In the branch by branch control strategy the jobs queue is used to hold oracles which have been sent by the Path Processors. The Oracle Generator is used for storage of oracles generated by the PPs and does participate in the creation of these oracles. The jobs queue contains the paths of all branches which have been discovered but not yet explored by the Path Processors. A PP always follows a set course through the choice points (for example, always picking the first choice) and sends the paths of all untried choices back to the Oracle Generator. The Oracle Generator then holds these untried paths in the jobs queue until a PP becomes idle and needs a new job. A particular oracle is then assigned to a PP by putting a host identifier into the job structure. This job information will wait on the distribution list until it is sent to the proper PP. Figure 6.1 shows the set of data structures needed to implement any of the non-backtracking strategies.

Job structures are the standard unit of information which must be relayed to a Path Processor. In general, a job structure consists of:

- An oracle describing the path to be followed.
- An identifier of the Path Processor to which this oracle goes.
- A command describing the function to be performed.
- The number of bits in the oracle (they are packed).
- Any extra information needed by the Path Processors.

When there are extraneous jobs which cannot immediately be assigned to a PP for exploration (as occurs in the bit by bit and branch by branch strategies), these jobs are put into the jobs queue. If there are no jobs on the queue when a PP has become idle and needs a new oracle to follow, the PP must wait on the *idle processors stack* until a job becomes available. This stack contains the unique identifier associated with each idle PP. Memory management is the reason for the *free list* data structure. This structure is an array containing the addresses of individual job structures which have been allocated memory space but are not currently being used. An initial number of job structures are allocated memory and placed into this free list. This reduces the number of calls to `malloc` which will need to be made during the execution of the Prolog program.

The last run array is used in the bit by bit strategy for control and by all of the Delphi strategies to provide fault tolerance. In the bit by bit strategy the Oracle Generator must provide the two single bit extensions to an oracle and place these new jobs on the jobs queue. The oracle to which the bits are appended is the last oracle explored by the PP which has become idle. Before an oracle is sent off to a PP for exploration, it must be saved in the last run array so that potential

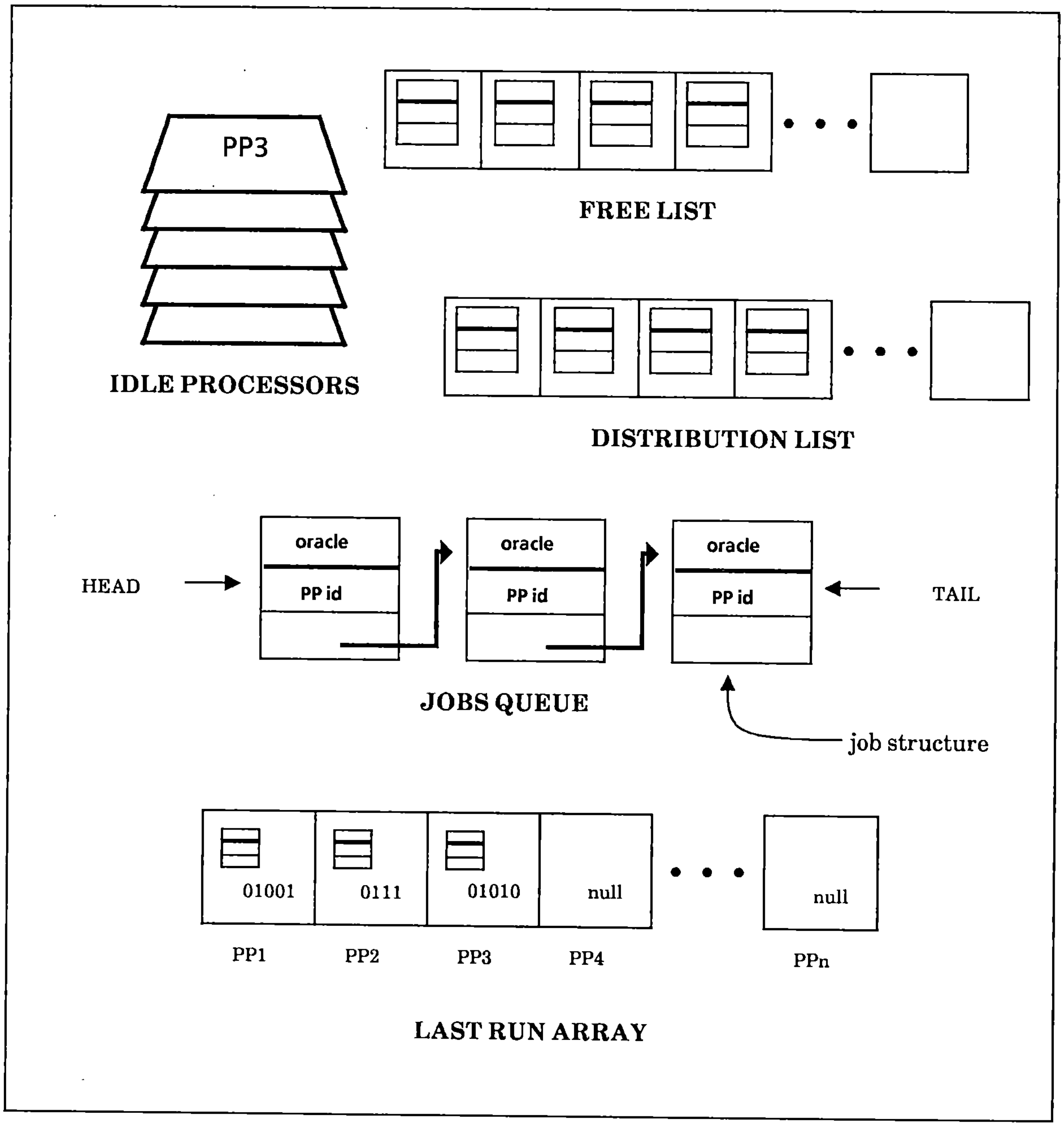


Figure 6.1 Data Structures of the Oracle Generator

extensions can be properly made. One of the features in Delphi which makes the system fault tolerant is the oracles contained in the last run array. If the host machine or an individual PP should crash during the execution of a Prolog program, the last run array will still be holding the oracles which were last sent to each of the failed PPs. Additional PPs can be initialised and given the oracles which should have been explored by the PPs which have died.

The function of the Jobs Distribution process is to communicate with the Prolog processes and send them their new oracles to follow. By supplying a separate distribution process, some of the burden of the Controller is alleviated. The jobs distribution process interrogates the Oracle Generator occasionally to pick up the entire queue of jobs waiting in the distribution list. After the distribution list has been collected, the Oracle Generator can turn back to the problems involved with maintaining the jobs queue and assigning oracles to idle PPs. The Jobs Distribution process will be busy communicating with the PPs and sending them new oracles to follow.

The final component of the divided Controller is the Answer and Exceptions server process. This process provides a means for global logging of the activities of all other processes. It is the only process which always functions as a server process. It never initiates communications to any of the other processes, it only receives information and transmits an acknowledgment.

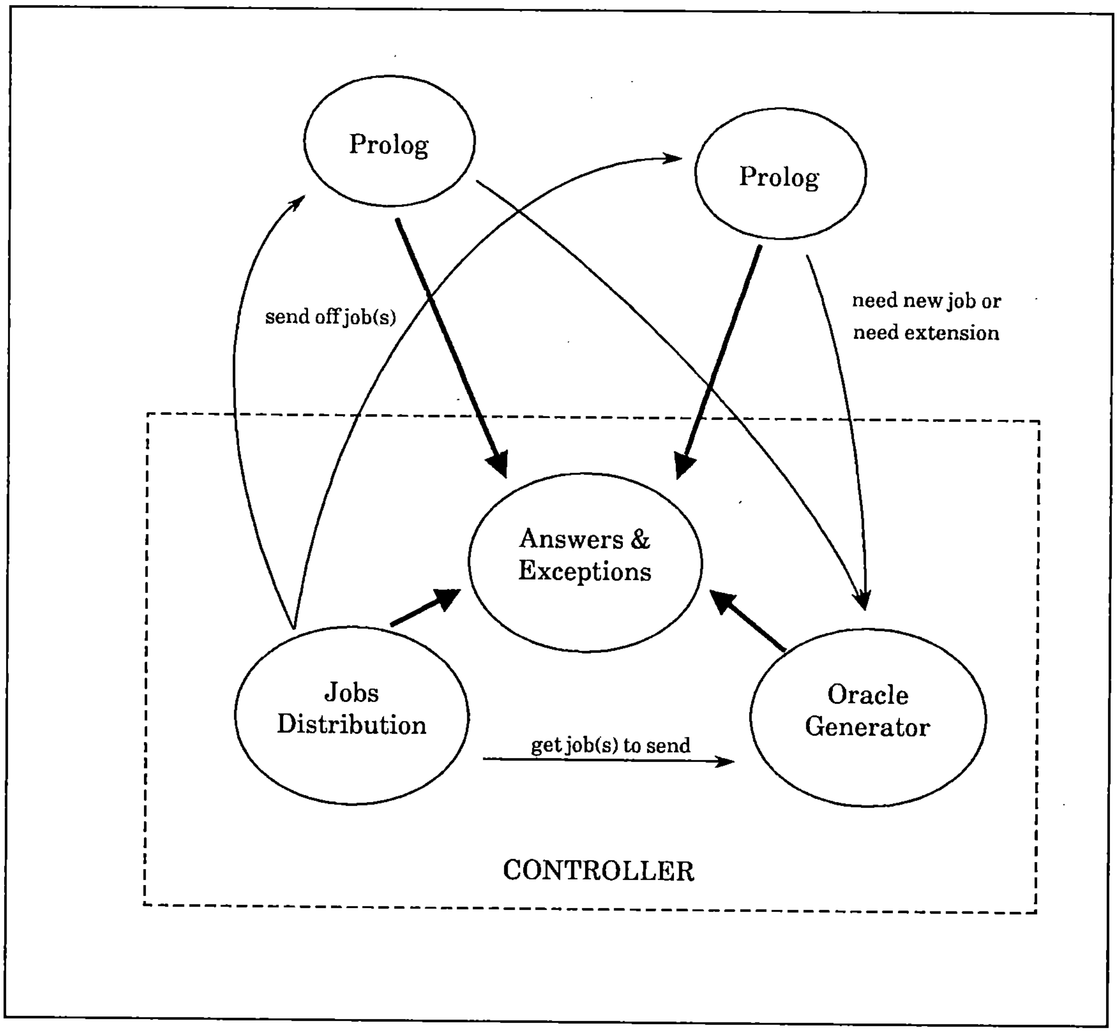


Figure 6.2 Individual Processes

The information it receives is usually a character string which needs to be logged. The message often consists of status information on a particular process, such as when they are initiated or halted. Solutions to the Prolog program which is being executed are sent from the PPs to the Answers and Exceptions server as *answer messages*. Faults which occur during the processing are also logged by this server process. The Answers and Exceptions server is not a mandatory process in the Delphi system. If it does not exist, then all of the logging information can be gathered from local log files on each of the host machines. A Delphi configuration consisting of two Prolog processes and the three separate process which comprise the Controller are shown in Figure 6.2.

Each of these individual processes is shown as a named ellipse. The direction of the communications among the processes is shown by the direction of the arrows. The Jobs Distribution process most often functions as a client by initiating communications to both the Oracle Generator and the Prolog processes. Only one command is sent to the Oracle Generator; the command to obtain all of the jobs waiting to be distributed among the Path Processors. These jobs are sent off to the various PPs with commands in the job structure to perform activities such as halting the execution of a PP or telling the PP to follow an oracle. The Prologs themselves function both as clients and servers at different times. When they have no work to do they initiate communications with the Oracle Generator to request a new job to be given to them. These processes then act as a server, blocking on input from their port and waiting for a new command from the Jobs Distributor.

To configure the processes shown in Figure 6.2, four communications ports would be created by the system. Each of the Prologs has one port on which to receive jobs from the Jobs Distribution process. The Answers and Exceptions process has a single port on which to receive messages from any of the other processes. The Oracle Generator has a port which is used both by the Prolog processes and the Jobs Distribution process. The Prolog processes request jobs on this port and the Jobs Distributor acquires the distribution list on it. The Jobs Distributor process mainly functions as a client, the Answers and Exceptions process mainly as a server, and the others function as both. These client and server processes can be run on any of the available host machines, either all together or distributed in some fashion among the machines.

Figure 6.3 shows two possible distributions of Delphi processes to host processors. Figure 6.3a is a simple configuration of three processes executing on a single host machine. The Answers and Exceptions server is not a mandatory process so this configuration represents the smallest amount of code needed to execute a Prolog program. A more common configuration of the Delphi machine is shown in Figure 6.3b. Each individual process is executed on its own host machine and multiple Prolog processes are configured. Most Delphi configurations using the Amoeba-transactions-under-UNIX IPC contain the Answers and Exceptions server for reasons of convenience.

The process of running a Delphi machine using the Amoeba-transactions-under-UNIX implementation is not user friendly. The code for each individual process must be distributed to the host machines and then each process must be manually started. The distribution can be performed automatically with a shell script using the `rcp` (remote file copy) or `ftp` (file transfer program) commands of ULTRIX. Initiating processes across a network is not accomplished as easily. In addition to initiating the client and servers, it is desirable to maintain a global status of all processes and be able to terminate them remotely. To help the user manage the Delphi clients

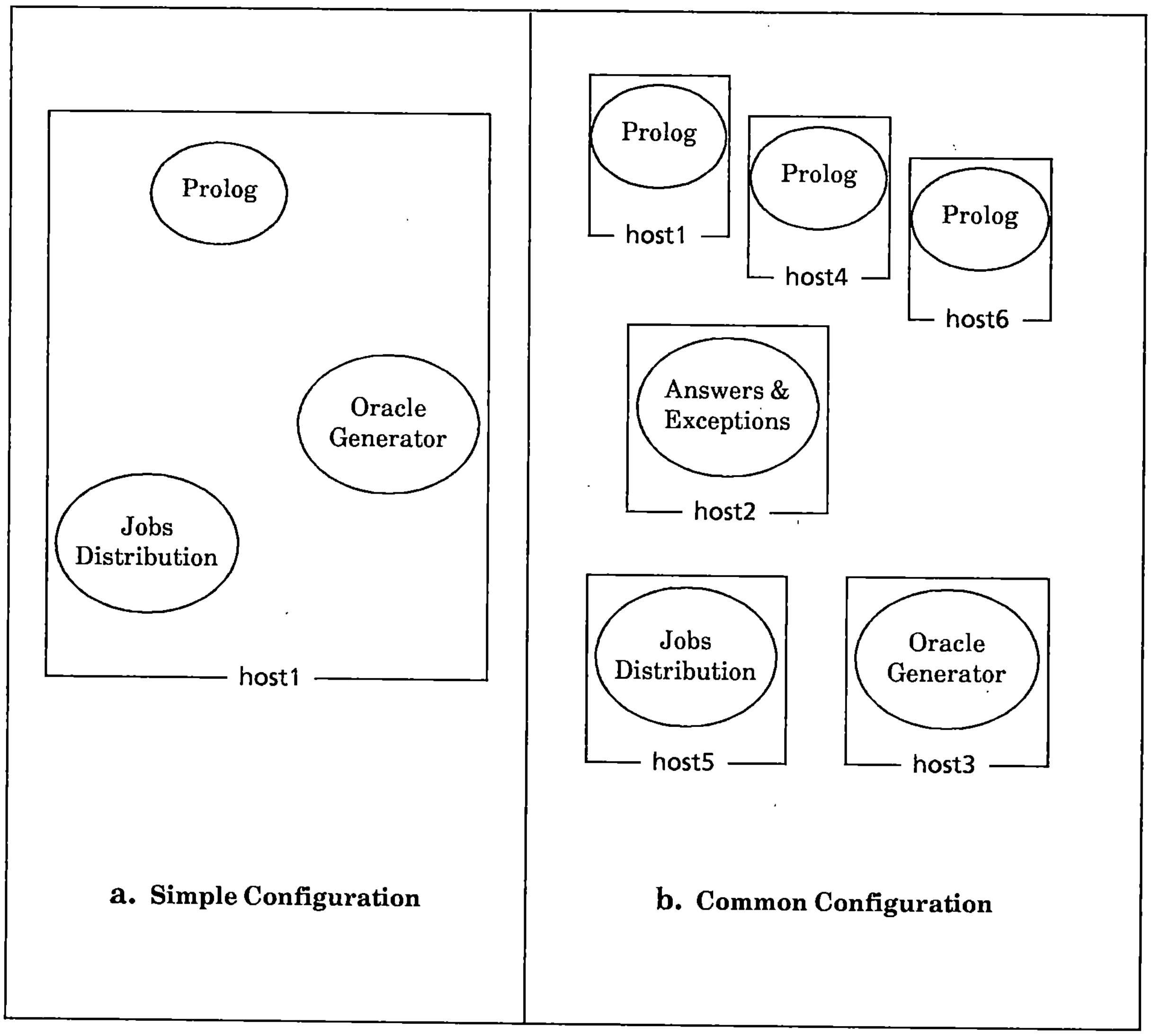


Figure 6.3 Two Configurations

and servers, an external *Process Management System* was developed. This system is external because it is not restricted to the management of Delphi processes. Any executable process can be distributed, started, watched and terminated by the Process Management System (PMS).

The PMS was designed to provide the convenience of using a menu driven program on a single processor which could initiate and watch all of the Delphi processes running on any of the host machines. This system was implemented using the Amoeba-transactions-under-UNIX IPC facilities and contains its own code for file distribution via Amoeba transactions. The PMS is self-contained code to help developers and users in debugging, initialisation and maintenance of a distributed system such as Delphi.

When Delphi is managed by the PMS, each of the Controller components is automatically initiated in turn. The Answers and Exceptions server is started first so that it can log the initiation of all other Delphi processes. The Oracle Generator is started, then the Jobs Distribution process and finally all of the Prologs are initiated in any order. As soon as a Prolog has been started, it immediately performs a transaction to the Oracle Generator to obtain a *Global Server Number*. All Prolog processes are equivalent so this number is needed by the other processes to identify each individual Prolog. This number remains with the Prolog process throughout its lifetime and is used for identification during a transaction.

After a Global Server Number is obtained, the Prolog process creates a unique port name for itself and announces its existence to the Answers and Exceptions server. The port name is a concatenation of the character 'P' and the Global Server Number so is guaranteed to unique for each Prolog process. After all Prologs have been initialised, execution of the Prolog program begins. Any solutions which are found by the Prologs are sent to the Answers and Exceptions server to be placed in the global log file. When the tree has been completely searched, the Prolog processes are halted. Each sends a message to the Answers and Exceptions server to "log out" and then terminates itself. At the end of a Delphi run the global log file (which is created on the same host machine on which the Answers and Exceptions server executes) contains the solutions to the Prolog program in addition to the logging messages for each of the Delphi processes.

Even with the PMS facilities, the Delphi implementation using Amoeba IPC was never as clean and easy to manage as the ULTRIX implementation. The problems originated with the Amoeba IPC mechanisms which were available when Delphi was implemented. The IPC forced a particular structure on the Delphi model causing it to be more complex than the model used for the ULTRIX implementation. The problems inherent in the Amoeba implementation will be discussed after a description of the process management tool (used by the ULTRIX implementation) known as the Internet daemon.

6.2 Internet Daemon

Imagine that all of the services associated with ULTRIX are constantly running in the background of a host machine waiting for clients to request their service. Figure 6.4 shows a few of the common services that might exist on such a system. Each of these individual services takes up time and space on the host processor and some of these services may rarely or never be requested by the users on the host machine.

A daemon can be described as a process which waits in the background for messages to be sent to it via its port connection. It is essentially an idle server process which becomes active when a client requests its service. In Figure 6.4 all of the services shown are daemon processes waiting for a connection to their port. When a request for a connection is made by the client process, the daemon receives the message through its port and acts on it accordingly. This model of multiple daemons is a great waste of resources. A socket connection is used up for each service available and memory has been allocated so that the process can run once it is awakened. Many of these services

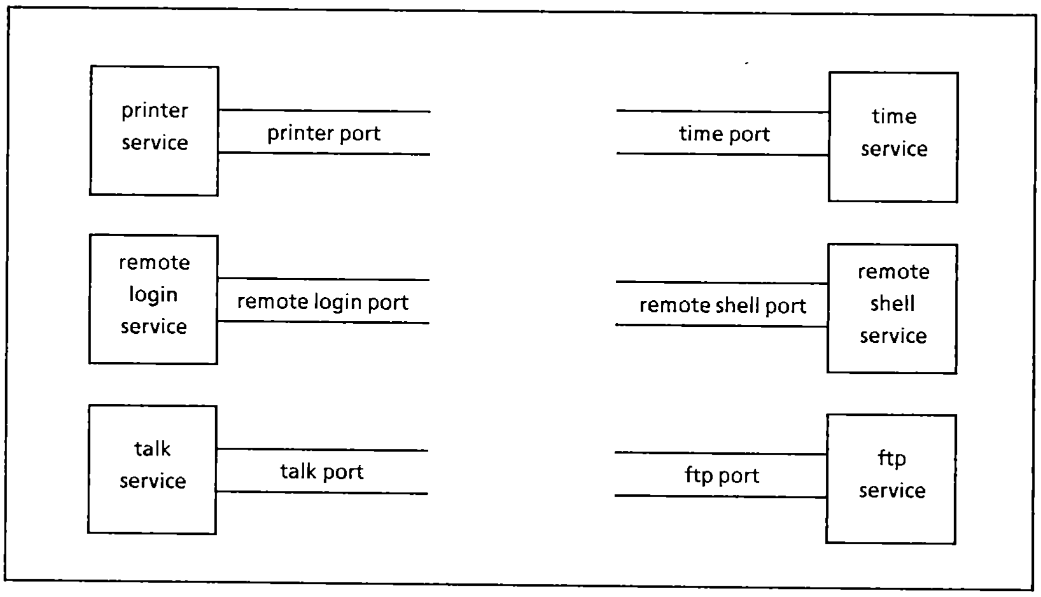


Figure 6.4 Multiple Services

may never be used by the customer, so the allocation of resources for them cannot be justified. The Internet daemon was designed to avoid some of the problems with having multiple services always available and running on a host machine [ARPA/Berkeley Services Reference Pages HP 9000 Series 300 1987].

The Internet daemon is a process which handles many of the services available on UNIX machines. It listens on numerous ports for requests to any of the services that it controls, and then acts as a temporary liaison between the client process and the service that the client wishes to contact. When a request is made to one of the Internet daemon services, the daemon receives the message, starts up the proper service process, and finally connects the client with the service it had originally specified. It is similar to a phone connection going through a switchboard where the Internet daemon is the switchboard. The beauty of this scheme is that only a single daemon takes up space on the host machine (see Figure 6.5). Other service processes do not exist until they are requested and become activated by the Internet daemon. This is a great savings on the resources of the machine.

How would a service such as the Internet daemon be created using the Amoeba IPC calls? A get request (`getreq`) performed by a server is a blocking primitive. Once a port name has been specified in the header, and the `getreq` has been performed, there is no way to alter the connection. The `getreq` will wait forever on that single port until a message arrives. What is desired is a method which allows more than one port to be tested to see if any messages have arrived on any of them. If one of the ports does contain a message, then the message can be answered by performing a `getreq` on that port. To test more then one port, a non-blocking transaction which reads a port must be allowed. The version of Amoeba which we used could not perform a non-blocking port test.

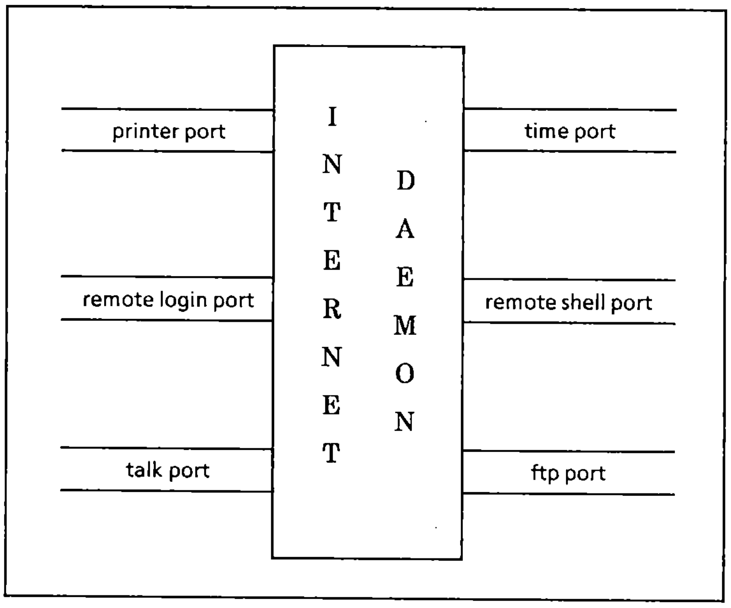


Figure 6.5 Internet Daemon Service

It would be difficult to use the Amoeba IPC to create a process similar to the Internet daemon. There is work underway to allow a set of ports to be tested using the Amoeba system, but no such facility was available at the time when this research was done.

Without a port select mechanism a constraint is placed on the types of communication models which can be implemented. For the Delphi system, there is typically one Controller process which receives messages from any of the Path Processors. A common model is one where a non-blocking port select mechanism is used. The Controller spends its time scanning the ports for incoming messages, answering requests from Path Processors which have sent a message. As this model could not easily be implemented with the Amoeba primitives a less intuitive protocol had to be created. The Oracle Generation Process waits for requests on one port only. All of the Path Processors write to this same port, and instead of the requests being queued up as they would be with a selection function, only one processor gets answered. If many PPs send messages simultaneously there is contention for the Oracle Generator port.

Figure 6.6 shows two models for port connections between the Controller and the PPs. In Figure 6.6a is a model with only one port on the Controller side. All of the PPs must communicate to the Controller through this single port. Figure 6.6b is a more desirable set of connections where the Controller has a connection for each individual Prolog Process. Maintaining a single port on the Controller side is not necessarily a negative feature. If the underlying communications system has been set up to handle contention, then with just one port the simultaneous requests from client processes could be queued and answered in turn. This would be no different from having multiple ports and scanning them sequentially for requests.

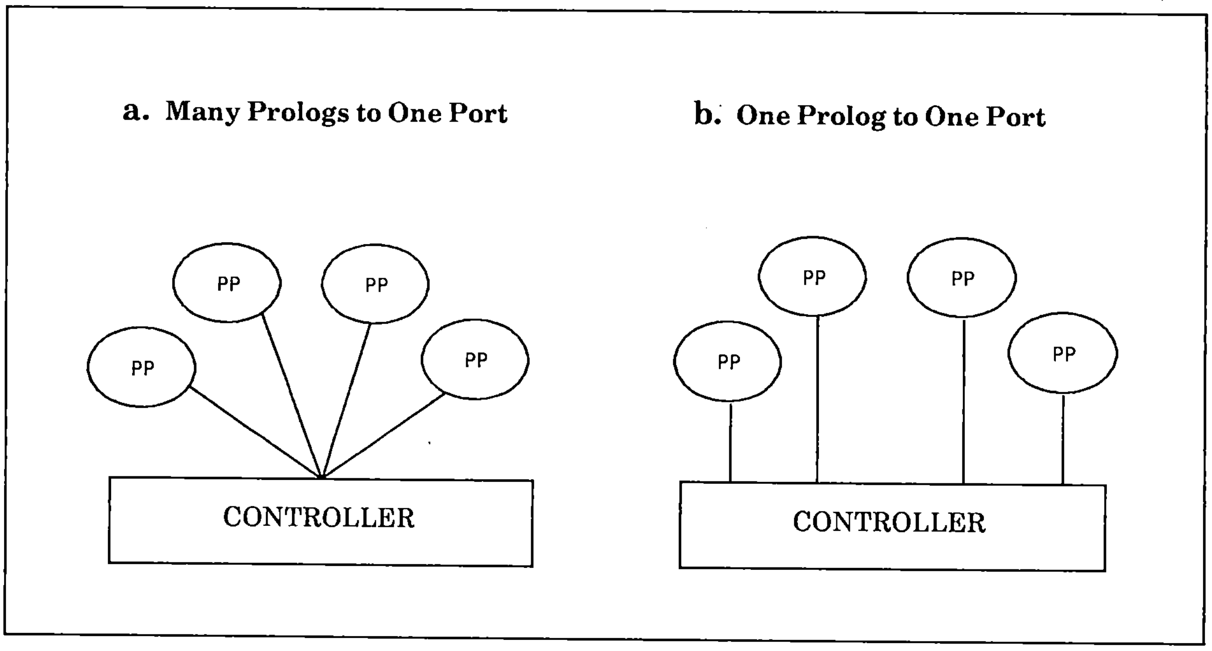


Figure 6.6 Models for Connections

The Amoeba system however, could not handle the contention properly. Multiple clients writing to the same server port usually caused the system to crash. Ironically, using the Amoeba IPC facilities in the Delphi machine forced an implementation where contention was common on the major port in the system. The Amoeba system could not handle this contention and responded by crashing when it took place. A widely tested IPC mechanism was chosen for the second implementation of the Delphi machine; the ULTRIX implementation of 4.2BSD IPC.

6.3 An Integrated Delphi Model

In this model of the Delphi machine, the Controller is a single process which performs all functions of the Oracle Generation Process, the Answers and Exceptions Server and the Jobs Distribution Process. No external process management system is needed to start processes on remote host machines or display the state of the network. This functionality has also been incorporated into the Controller of the integrated Delphi model. The implementation created from this model uses the 4.2BSD IPC facilities implemented within the ULTRIX operating system. This second implementation of the Delphi machine will be called the ULTRIX implementation.

With the ULTRIX implementation, the user does not have to understand clients, servers or distributed systems to execute a Prolog program on multiple host machines. A source file can be written and a single command invoked to take this file through the compilation, distribution, loading and execution phases of Delphi. A global log file will be created on the host machine where the initiation command was invoked. Any solutions found along with all of the logging messages sent throughout the Delphi run will be contained in this file. An example of a global log file can be found in Appendix 6a.

If a user wants to avoid the compilation phase of execution on the Delphi machine (perhaps the same compiled code is to be executed more than once) or change the parameters for a control strategy, only a single file needs to be altered. Facilities have been provided to make Delphi simple for those willing to use the default Delphi setup. If the user is interested in tailoring the system, (for example, to use particular host machines on the network in preference to others) then more has to be learned about the wide range of tools, methods of initiation, and various configuration files which are an integral part of Delphi.

For benchmarking purposes, the distribution and execution phases were mainly used. This involves distributing all of the pertinent files to the Path Processors and then invoking the command `Controller` with any command line arguments. A shell script controls the distribution of files using the minimal number of remote copies to ensure that all of the files are always up to date. The functions performed by this shell script include:

- Checking to see which of the potential host machines are available and running.
- If a host machine is available and is to be included in a Delphi run, check that all of the files needed are up to date on that host and in the proper directories.
- If a file does not exist on the host machine, or is an old version, copy it to that machine.
- Create log files and report any errors or crashed machines.

The first function gives an overall view of the state of the network; which machines are available and which are not. This is not necessary for the functioning of Delphi since if a machine is down at initialisation time, that machine is just not used and an alternate is found if possible. It is mainly of interest to the user, and especially important when benchmarking, to see how many machines are available for a run.

The method of checking for the existence of necessary files uses a file distribution and configuration file. All files named in this configuration are maintained on a source machine. Equality between these files on the source machine and on each of the host machines is checked. If a source file has been modified since it was last copied to the host machines, it is updated. If any of the source files do not exist on the host machine, they are copied over to that host.

Deletion routines for particular file types can also be placed in the distribution configuration file. These routines are useful since numerous log files are created and disk space on the host machines might become full if they were not occasionally deleted. For networks running `nfs` (network file system) the distribute command is not needed as all host machines appear to use the same file server. Because some of the machines used during this research did not have facilities for remote file servers, the distribute phase of Delphi was necessary.

Files to be distributed or maintained on all of the Path Processors include:

(1) Prolog master server daemon
(2) Prolog run time system
(3) Prolog system files
(4) Compiled source file to be executed
(4) The file containing the standard input to the Prolog systems

The Prolog run time system is written in the C programming language. This is the code which executes and controls the low level WAMO (Warren Abstract Machine with Oracles) instructions. All of the communications code and low level interfaces to the control strategies and the Controller are also a part of the Prolog run time system. When this system is executed, it immediately loads all of the Prolog system files which is that part of the run time system written in Prolog. These files include all of interfaces between the C code and the user commands written in Prolog. As an example, there is a Prolog command which allows the user to interactively change control strategies. This command is the interface to the C code which actually implements the control strategy change. After all of the Prolog system files are loaded into the program space, the source code translated with the Delphi compiler is loaded using the special Delphi loader.

The executable command which starts the Prolog run time system will be called `prolog`. Depending on how this executable command is initiated determines whether the Prolog master daemon or the standard input files are used at all. If the `prolog` command is initiated on a terminal, then the session will be in some way interactive. The user gets the choice as to whether the system will be run *stand alone* or as one of a network of Path Processors functioning as a Delphi machine. If the `prolog` command is started by the Prolog master server daemon, then it must be connected to the Controller and the only option is a networked system.

As the Delphi machine is intended to be run on a large number of processors, it is necessary to provide a high degree of automation in starting the Path Processors. For debugging purposes (or just to quickly test out some Prolog code) it is also useful to have a stand alone system without any network connections. If the `prolog` command is given at a terminal, then the user chooses whether this Prolog process will be stand alone or one of the networked Prologs of a Delphi machine. Upon giving the command `prolog` the user must supply an answer to the following query:

```
service for direct Prolog connection?
```

This query only occurs when the command has been initiated from a terminal. It is also possible (and in most setups, this is standard) that the `prolog` command is initiated by a daemon process. When `prolog` is initiated by the user, the query determines whether what is required is actually a stand alone Prolog system or whether connections to the Controller are to be made manually. Two possible stand alone configurations and three levels of initiating networked Prologs, have been incorporated into Delphi.

Executing `prolog` at a terminal and answering the service query provides different characteristics for standard input and standard output files. Table 6.1 shows the set of parameters for standard input and output according to how the query is answered.

Answer to query	Standard input	Standard output	Standard error	Prolog setup
NONE	terminal	terminal	terminal	stand alone
STDIN	`prolog_standard_input`	terminal	terminal	stand alone
service name	socket	socket	socket	networked

Table 6.1 Terminal Setups

Answering the service request with 'NONE' states that no service (port connection) is to be used for this run. The Prolog process functions with input and output directed to the user (terminal). This is the standard interactive mode similar in function to many sequential Prologs. When 'STDIN' is the response, the standard input is read from the file `prolog_standard_input` with output going to the terminal. This is a useful method of starting `prolog` if many initialisations take place; they will automatically be read from a file. Stand alone does not only mean running on a single processor—it means that the Prolog has no interaction with the Controller process. In contrast, a Delphi configuration consisting of only one Path Processor can run on a single machine, but will still consist of a Controller and a Path Processor with a socket connection between them. Any other response besides 'NONE' or 'STDIN' assumes that the response is a service name. The Prolog process will then wait on that port name for a socket connection to be established from the Controller.

6.4 Initiation of Processes over the Network

Three methods of Prolog process initiation have been provided to allow the maximum flexibility for starting a networking Delphi session. A networked Delphi session is one where all of the Prolog processes are connected by a socket to the Controller. This socket connection functions as the standard input, standard output, and standard error streams for the lifetime of the Prolog process. There is no user interaction at all with any of the Prolog processes once the connection to the Controller has been established. The methods of initiating the Prolog process, and allocating a socket connection or port for each Prolog range from a *direct Prolog connection* to *automatic start up* using the Internet daemon. The goal of any of the process initiation methods is to automatically start each Prolog on its proper host machine and establish a connection for each of them. A single connection between the Controller and each of the Prolog processes is the only communications channel that is necessary. Figure 6.7 is an example Delphi configuration consisting of eight Prolog processes distributed over five host machines.

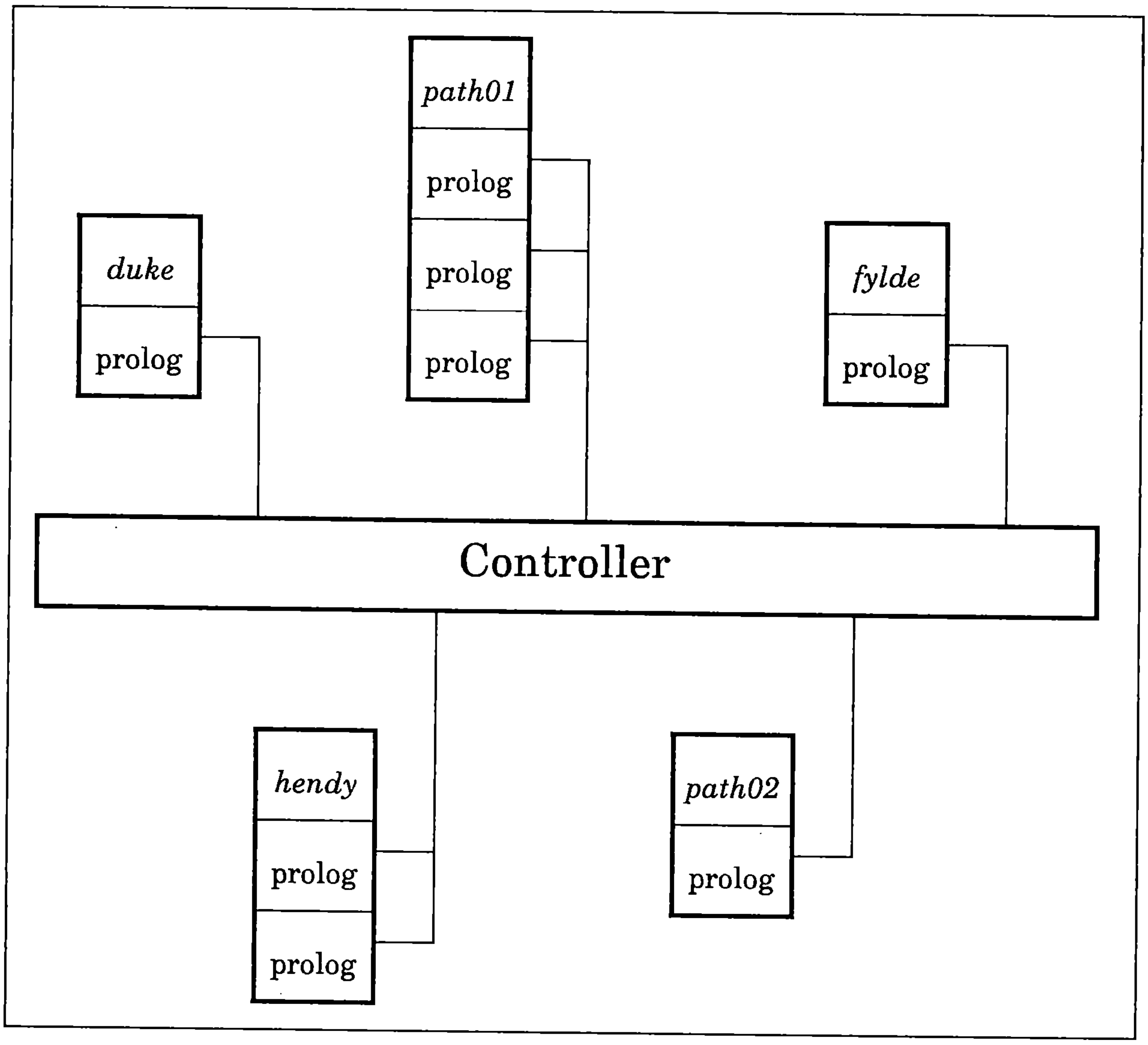


Figure 6.7 Minimal Connections

In Figure 6.7 each rectangular box with a bold-faced outline represents a separate host machine. The name of the host is shown in italics at the top of each box. No host name has been given for the Controller process, but it can be assumed that it is a different host from any of the other named machines. The Controller could be run on any of the host machines including those which run Prolog processes. Most often, if enough host machines were available, a separate host machine was used to run the Controller. Figure 6.7 demonstrates the minimal number of connections that a Delphi configuration with eight participating Prologs can have. Each Prolog has an individual socket connection to the Controller even if they are executing on the same host machine. The minimum number of connections to the Controller for a Delphi configuration is equal to the number of Prolog processes. These connections are sockets or logical connections and are not related to the physical connections between the host machines.

The physical connections for the machines used in the Delphi research are shown in Figure 6.8. The Tower and the Old Music School (OMS) are two sites at the University of Cambridge. Each site has its own Ethernet and there is an optical fibre link between them.

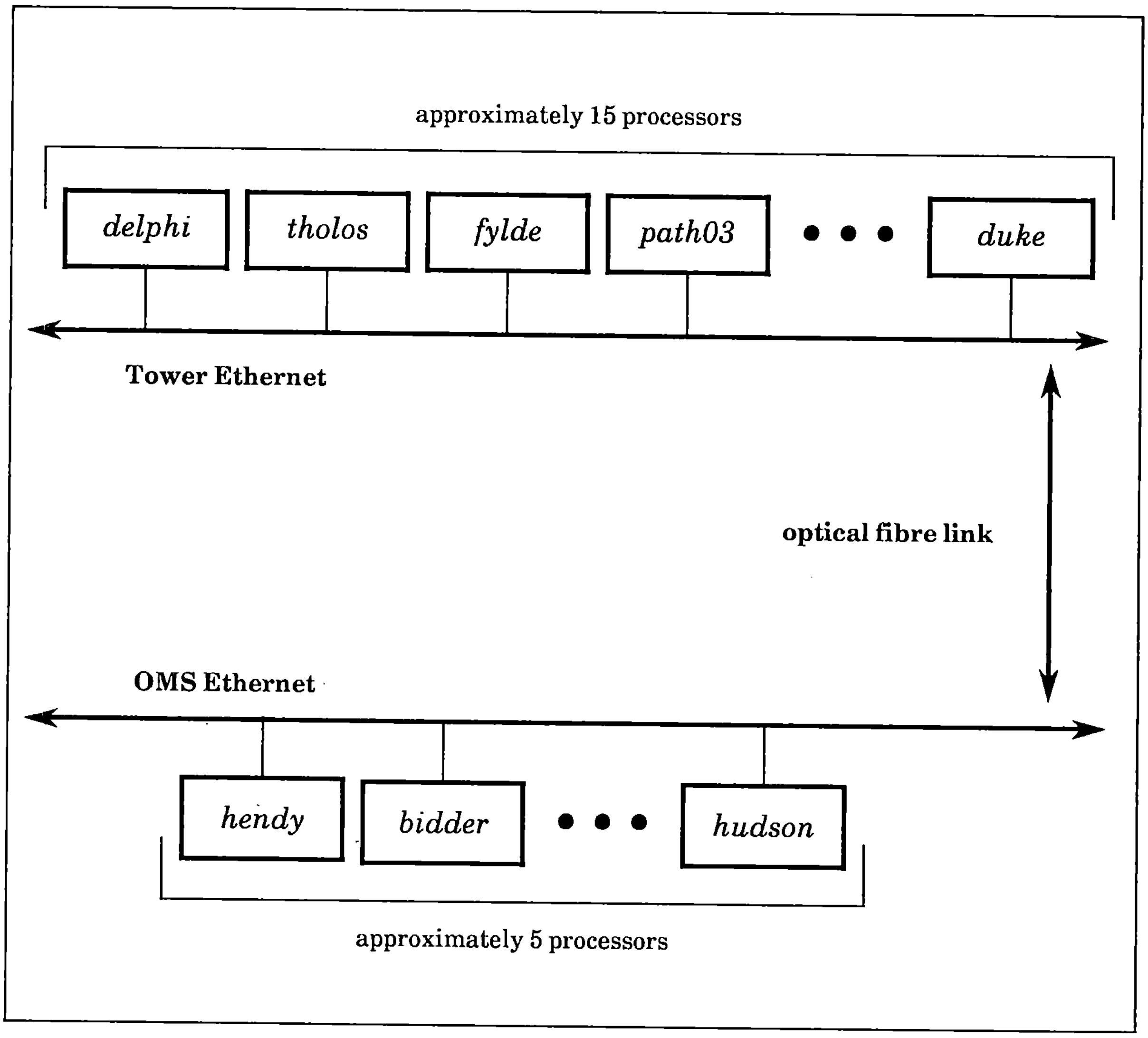


Figure 6.8 Physical Connections

To create the example configuration shown in Figure 6.7, it is necessary to access each of the host machines and start some number of Prolog processes running on them. There are three levels of mechanisms provided within the Delphi system for initiating Prolog processes over the network.

The levels are associated with the number of intermediate processes which are created to aid in the final goal of initiating Prologs. At increasingly higher levels of abstraction the details of process initiation over the network are hidden from the user. From the user's point of view these levels range from having to manually start each Prolog (direct connection) to an automatic initiation through the use of a configuration file (Internet daemon connection). Table 6.2 shows four initiation methods in relation to the user's involvement and the number of processes which are created.

name	**Direct**	**Master**	**Daemon**	**Internet Daemon**
level of abstraction	0	1	2	2
user interaction	for each Prolog	for each host machine	only when a machine crashes	never
number of DelPhi initiated processes *(excluding Prologs)*	0	1	2	1

Table 6.2 Levels of Prolog Initiation

The direct Prolog connection is at the lowest level of abstraction, level zero. Each Prolog system is individually initiated by the user with no automatic control at all. The user must log onto each host machine, start the Prolog(s) and supply a service port name for each of them. Any number of Prologs can be started on any of the host machines so this initiation method can quickly become tiresome. In addition to the Prologs needing initiation, the Controller process must be started on some host machine. When the Controller is executed, it runs a menu driven program which can complete the direct connections to the Prolog processes. Appendix 6b shows example sessions with the Delphi system where each of these mechanisms is demonstrated. Level zero interaction is very similar to running the Amoeba implementation without any aid from the external process management system. Each process must be started individually by logging into each of the host machines in turn, and initiating each of the processes to be run on that machine.

The level one initiation method is named Master because it uses a Prolog Master server process to spawn Prolog processes. The Master initiation method allows the user to view a configuration as a host by host initialisation procedure instead of the Prolog by Prolog method at level zero. At this level, a Master process controls the initialisation of all Prologs on that host machine. Instead of the user having to log onto each of the host machines and start up each individual Prolog on that machine, only the Master process now needs to be started. Still the user must log onto all of the host machines, but only one process, the Master, needs to be initialised per host machine. At level zero, the user started the Prologs individually. At level one, the user starts a Master process which starts the Prologs.

The highest level of abstraction, level two, allows the user to look at process initialisation on a configuration by configuration basis. A Delphi daemon process is left running on each of the host machines. This daemon controls the initialisation of the Master process, and this Master controls the initialisation of each of the Prolog processes. There is now no need for the user to have to log onto each host machine to initiate a new Delphi configuration. The configuration itself is automatically read from a file with no user interaction at all.

Also at level two is an initiation method which occurs through the use of the Internet daemon. This method is just a simplification of the Delphi daemon method described above. The difference being that instead of a Delphi daemon running on each host machine, the ULTRIX maintained Internet daemon is used instead. The functioning of each of these daemons in relation to the Delphi machine is equivalent; they both initiate the Master process on each host machine. However, the Internet daemon is automatically initialised with each reboot of the processor, whereas the Delphi daemon needs to be manually restarted. When around twenty machines are involved in a particular Delphi configuration, this automatic initialisation is an improvement. At the Computer Laboratory, it is not uncommon for each of these machines to be rebooted more than once a day. With the Internet daemon listening for Delphi configuration requests there is no reason for the user to have to log onto any of the participating host machines at all. As a summary, Figure 6.9 shows the hierarchical nature of Prolog process initialisation at each of the three levels described.

6.5 Multiple Users or Configurations

All of the configurations have been explained as if only a single user were running the Delphi machine. The process initiation methods described allow multiple users to run Delphi configurations, or a single user to run more than one configuration simultaneously. Configuration is intended to mean the setup of Delphi processes on each of the host machines which includes a single Controller process. It is most common to start a new configuration for each Prolog problem that is to be solved. Often the problem involves loading a single compiled Prolog program with one or more top level queries. It is however possible, just as it is in standard sequential Prologs, to have a single Delphi session read and compile multiple source files and answer multiple queries.

A configuration can be thought of as a Prolog session starting from initialisation of processes by the Controller, and ending when the Controller is exited. It is possible that a single user may

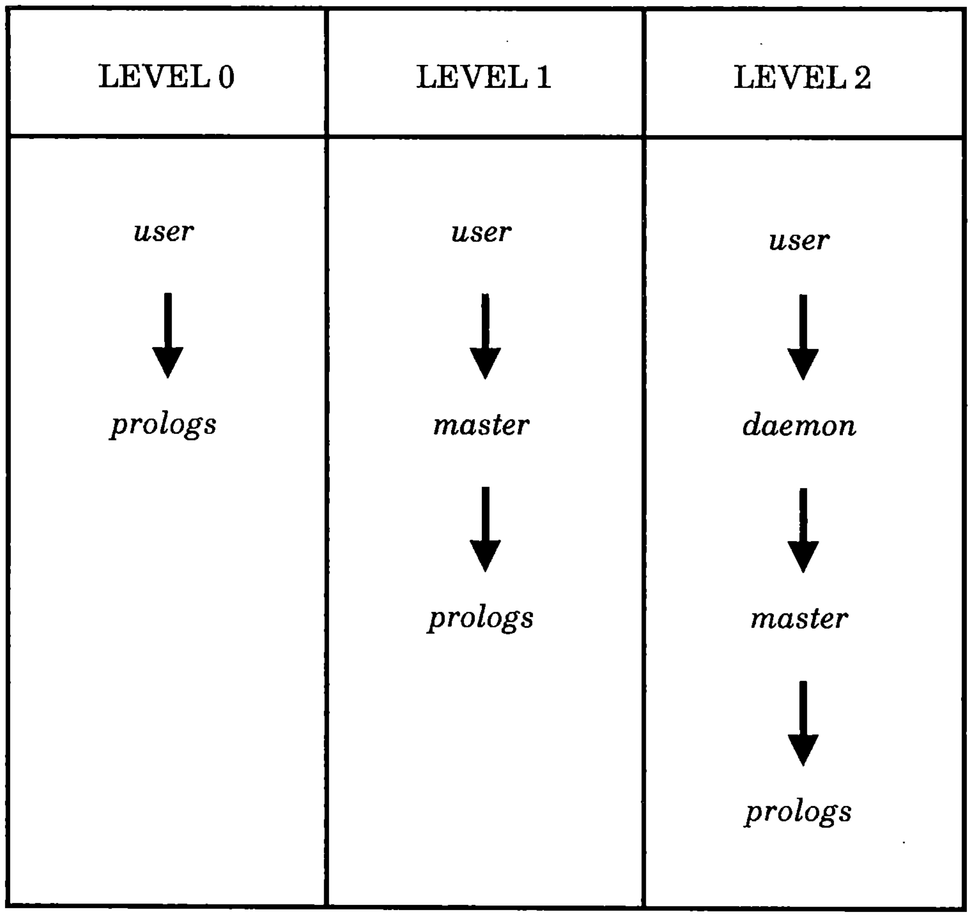


Figure 6.9 Process Initiation Hierarchy

want to run more than one configuration or session simultaneously, perhaps because the first session is going to take a long time to complete, and a separate Prolog program also needs to be executed. Also, multiple users may be running different Prolog sessions on the same or different host machines. The process initiation methods which have been described will handle all of these eventualities.

Initiation levels zero and one are very similar in a description of how individual users would go about starting their own Delphi sessions without interference from one another. With level zero, every Prolog process started by an individual, called User1, has the privileges and ownership associated with the account under which User1 is operating. If a second user, User2, starts Prolog processes on any machines, these processes have the same privileges associated with, and are run within User2's account. Since all processes (excepting the daemons) run under the user's account, there are no name clashes on any of the host machines. There is no problem with user specified port assignments as each user must keep track of the set of ports that he uses. Named ports (actually any ports) on a host machine can each be used only once. A second user trying to use a port name which has already been assigned will get an error message "port in use".

This same argument applies when starting Master processes (level one method of initiation) which are activated by multiple users. These processes run under the user's account and so have no identity problems. If a user wants to run multiple sessions (at the zero level of abstraction), it is entirely up to that user how the Prolog processes are connected together to form the various configurations. With level one, the multiple Master processes that run on each host are completely independent processes which spawn independent Prolog processes. The port names for the Prologs are assigned by the Master processes, so the user is spared the effort of remembering which ports are associated with which Prologs, and which Prologs are associated with which session.

The level two initiation mechanism was specifically designed to handle multiple configurations and multiple users. Before describing how the level two initiation procedures work, we shall take a look at what a single user configuration looks like on a particular host machine. Figure 6.10 shows a single user configuration, using a level two process initiation procedure, for a single host machine.

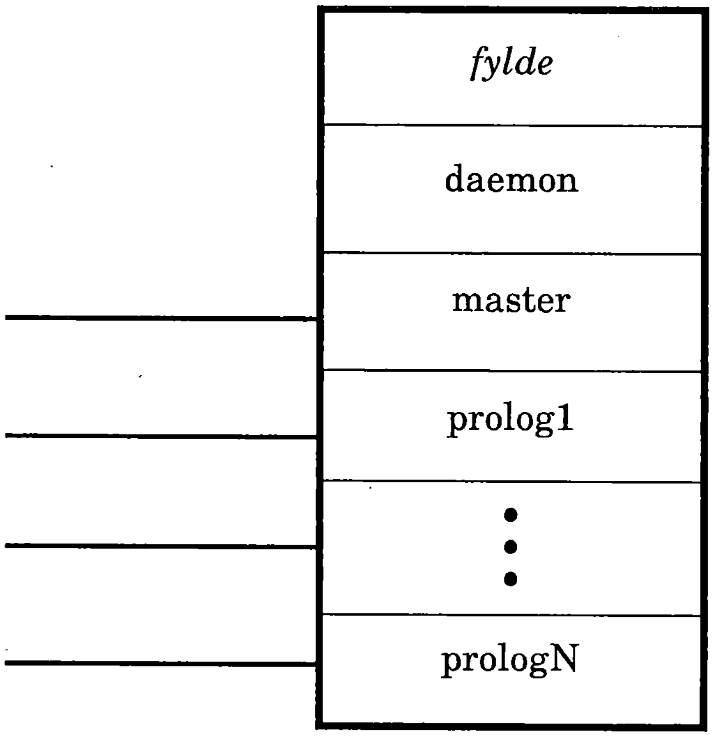


Figure 6.10 Single User Configuration on a Single Processor

The host machine's name is in italics at the top of the rectangle. All of the rest of the boxes contain individual Delphi processes which are run on the host. For level two, a daemon process needs to be running; this daemon can represent either the Delphi daemon or the Internet daemon. The Delphi daemon was mainly used during the testing phase of this initiation method. The functioning of this process has been completely taken over by the Internet daemon. One Master process is started by the daemon, and this Master initiates all of the Prologs required. The rest of the boxes have individual Prolog processes in them showing a minimum number of Prologs equal to one. There can be zero Prolog processes running on a host machine, but in that case the extraneous hosts (along with their daemon processes which continue to run) will not be shown.

There is a communications link to the Controller for each of the Prolog processes on the host machine, plus one additional connection for the Master process (see Figure 6.10). It is not absolutely necessary to maintain the connections to the Master processes after the Prologs have been initiated and their connections to the Controller established. If this connection is maintained, it is easy to initiate additional Prolog processes if they are required throughout the lifetime of the configuration. Most of the control strategies have been organised so that additional Prologs added at any time during the computation will be used. Figure 6.11 shows the Delphi processes on the same host machine when multiple users, multiple sessions or both are being run.

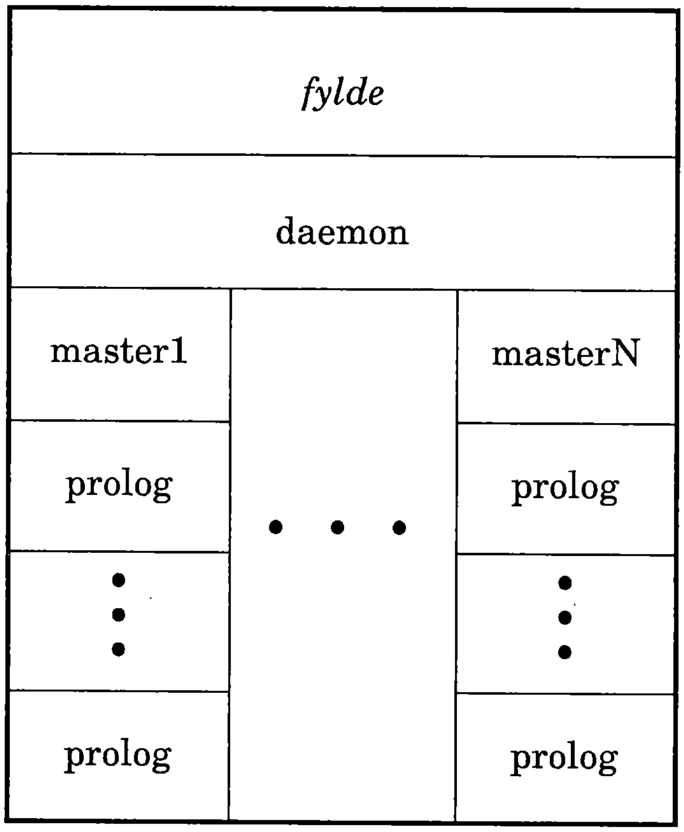


Figure 6.11 Multiple User Configuration on a Single Processor

Each different configuration (whether it is the same or a different user) requires that a new instance of the Master code to be run. The Master initiates a certain number of Prolog processes and automatically establishes unique port connections between these Prologs and the Controller.

Separate Masters initiate separate Prologs, so the processes concerned with a particular configuration never interfere with any other configurations. Notice that the daemon process is shared between all users and configurations.

Once the daemon process has been started, it is impossible for any other instance of a Delphi daemon to be initiated on the same host machine. This is because no port parameters are given when the daemon is started; this process always uses the special port named 'delphi'. If the Internet daemon is functioning on any of the host machines it is unnecessary and impossible to start the Delphi daemon process at all. This is because the Internet daemon also listens for port connections on the port named 'delphi', so the Delphi daemon process would not be able to bind to this already assigned 'delphi' port.

As an example of the ease in using the automatic start up facilities of Delphi, the initialisation command and the configuration files are shown for two users simultaneously running sessions. User1 has started his Delphi session by executing the command `Controller 5` with a configuration file:

```
path01      3
path02
duke
fylde
hendy       2
hythe
tholos
```

`Controller` is the command name which initiates the Controller process with the level two initiation method. On the command line, the first parameter means to create a session from the first five host machines in the configuration file, and then begin execution. The configuration file does not contain a one when there is only one Prolog process to be run on the host machine; one is the default number. As no particular problem file has been listed on the command line, each of the Prologs begins executing using the default file `prolog_standard_input` as the standard input for the Prolog processes. User2 has issued the command `Controller 4 myfile` and User2's configuration file contained the following lines:

```
duke
hendy
path01      3
path03      2
path00
laira
```

Here the user has specified the file to be used as standard input to the Prolog system (this file contains the parameters to specify both a program to execute, and any other setup features for the Prolog system) as file `myfile`. The first four host machines in this user's configuration file are used in the session. Figure 6.12 shows how the Delphi processes would appear after both User1 and User2 had issued their respective commands.

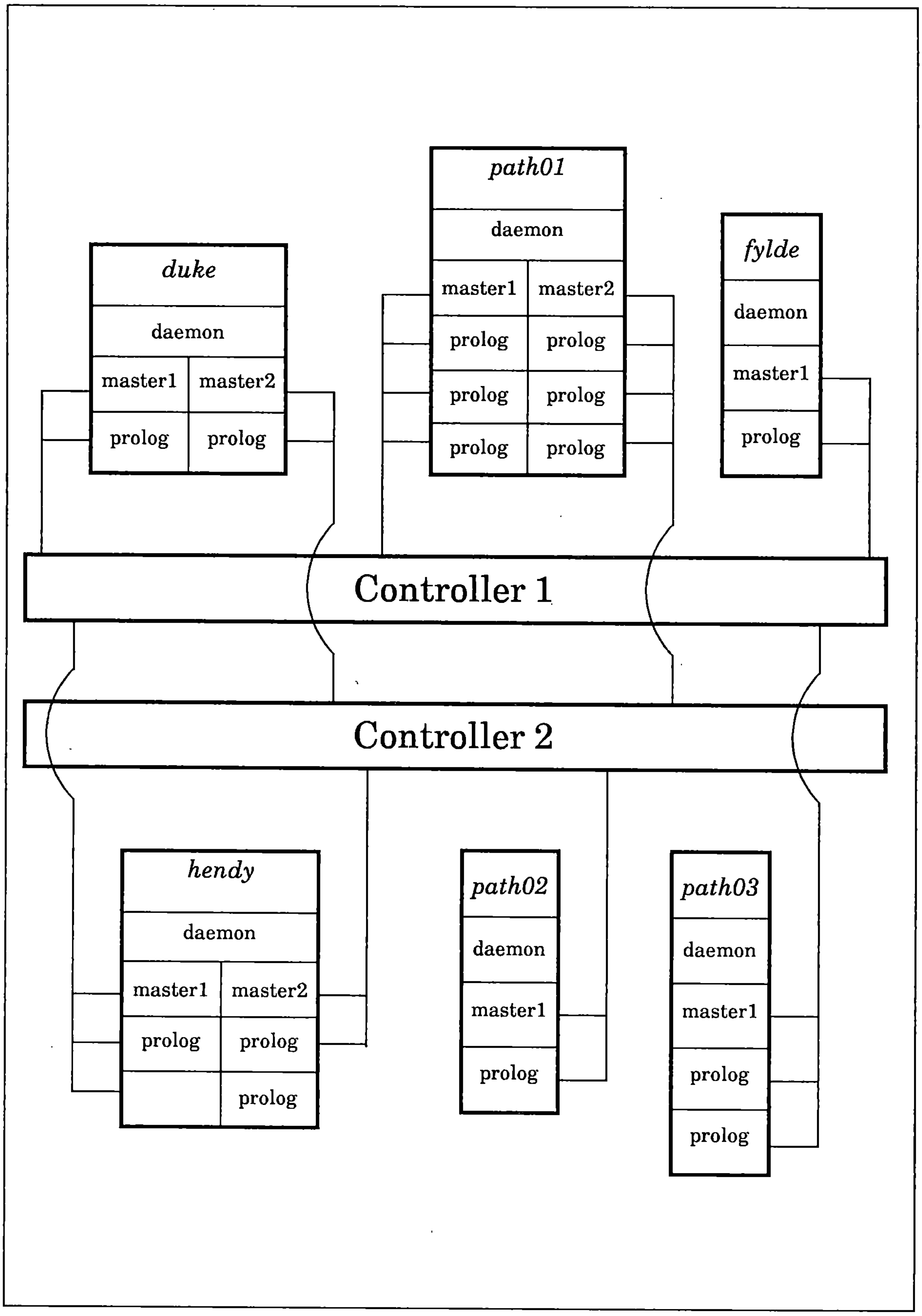


Figure 6.12 Example Delphi Configuration with Two Users

Chapter 7 Control Strategies

Each of the control strategies described in this chapter makes use of one or both of the following techniques:

- The use of oracles to recreate an environment (all strategies).
- Tree partitioning algorithms which divide the work of exploring a search space among multiple processors (backtracking strategies).

Both the use of oracles and tree partitioning strategies have been independently the focus of other research. In Chapter 2, Section 2.5, research focusing on the application of oracles to the creation of parallel Prolog models was described. Section 7.1 supplements Section 2.5 by relating the control strategies described in this chapter to the area of tree search algorithms.

7.1 Categories of Control Strategies

Searching techniques for AND/OR trees have been developed by researchers working in a number of disciplines including: game playing, theorem proving, planning, expert systems and Logic Programming languages[1]. To improve the computational efficiency of these tree searches parallel formulations of the algorithms have been proposed[2]. The Delphi control strategies described in this chapter are related both to previous work in the area of parallel Logic Programming languages [see Chapter 2] and to tree partitioning algorithms developed for game playing programs[3]. This section provides a closer look at the tree searching algorithms related to the Delphi control strategies.

Figure 7.1 shows the categories of control strategies which have been investigated during this research. Only exhaustive search strategies have been investigated as the purpose of this research is to exploit OR-parallelism in nondeterministic problems. The category of exhaustive search can be divided into two major types; the backtracking and non-backtracking control strategies. It is this division which defines the relationship of the control strategies to previous research. The non-backtracking strategies are related to previous work on the Delphi machine [Clocksin 1987,

1 Rao, V.N., and Kumar, V., Parallel Depth First Search. Part I. Implementation. *International Journal of Parallel Programming*, Vol. 16, No. 6, 1987.

2 Li, G. and Wah, B.W., Computational Efficiency of Parallel Combinatorial OR-tree searches. *IEEE Transactions on Software Engineering*, Vol. 16, No. I, January, 1990.

3 Finkel, R., and Fishburn, J., Parallelism in alpha-beta search. *Artificial Intelligence*, Vol. 19 pp.89-106, 1982.
Akl, S.G., Barnard, D.T., and Doran, R.J., Design, Analysis, and Implementation of a Parallel Tree Search Algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-4, No. 2, March 1982.
Newborn, M., Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 10, No. 5, September 1988.

Clocksin and Alshawi 1988, Alshawi and Moran 1988], while the backtracking strategies resemble tree partitioning algorithms[3].

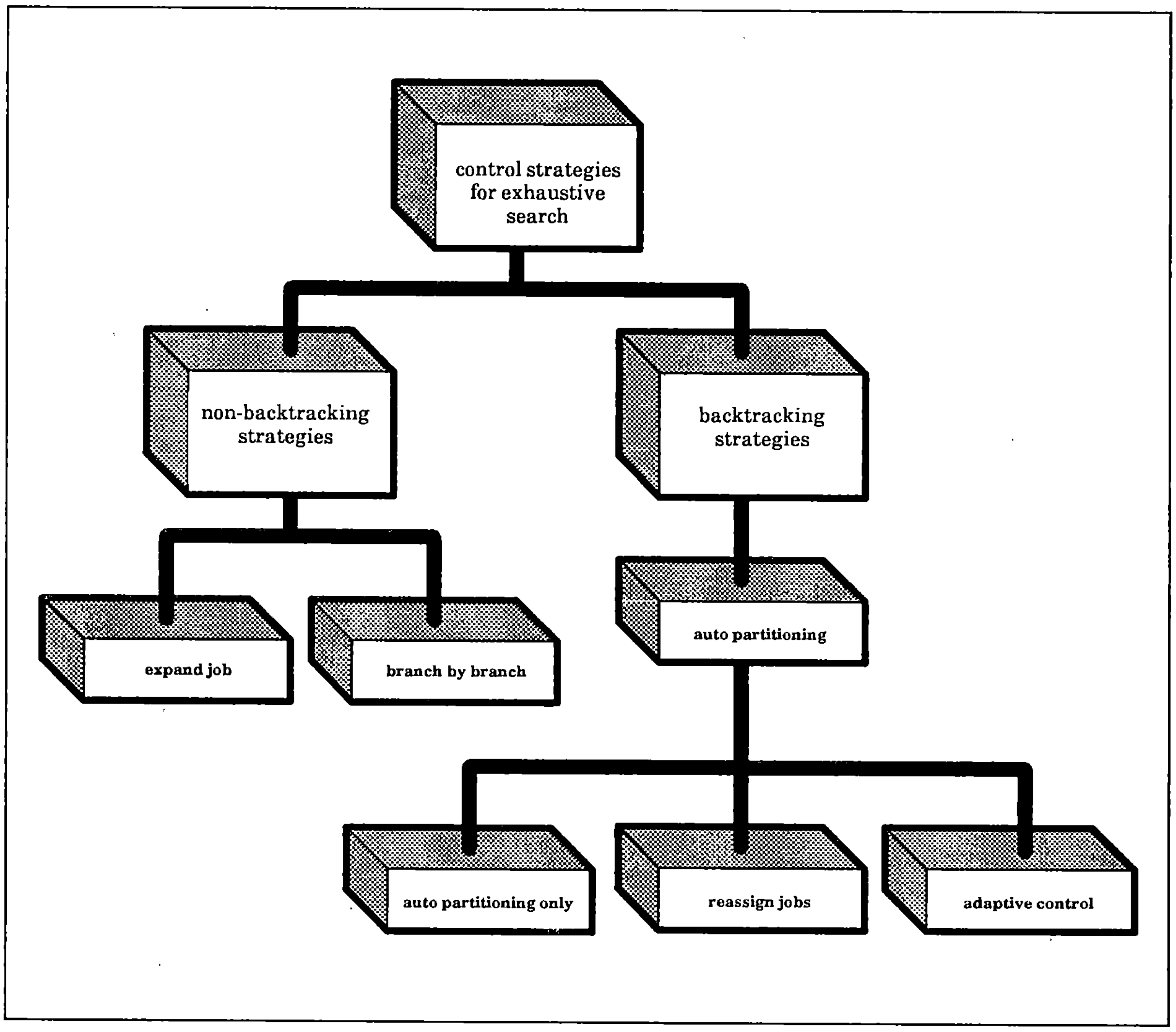


Figure 7.1 Control Strategies

Within the non-backtracking strategies two types of control are identified; the *expand job* and the *branch by branch* strategies. *Expand job* strategies are those which have some sort of prefix string or oracle to which extensions are added. Permutations of these extensions are generated creating new oracles to follow. The expand job strategies described in this chapter (Section 7.5) are equivalent to those proposed in Clocksin and Alshawi [1988]. The *branch by branch* strategies demonstrate the minimum number of communications which must occur in a non-backtracking strategy. *Branch by branch* is an original strategy which was not proposed in any of the previous Delphi research containing non-backtracking strategies [Clocksin 1987, Clocksin and Alshawi 1988]. All of the previous work on the Delphi machine [Clocksin 1987, Clocksin and Alshawi 1988, Alshawi and Moran 1988], and related research using oracles [Shapiro 1989] contain algorithms which have a strong similarity to the non-backtracking algorithms described in this research. The similarity is that the generation and following of oracles is the method for dividing the search space among multiple processors. In the backtracking strategies described in this chapter (Sections 7.7 - 7.9), oracles are used, but not for the division of the search space.

Backtracking control strategies (described in this research) all begin with an initial strategy or mode called *automatic partitioning*. Automatic partitioning allows the processors to partition the search space (from the root of the tree) without any communication to the Controller. After the initial automatic partitioning mode, each strategy executes in accordance with its individual mode of control. Three generic categories of backtracking strategies are shown; *automatic partitioning only*, *reassign jobs* and *adaptive control*. With *automatic partitioning only*, there is no extra control after the initial automatic partitioning mode has been completed. The search space is automatically partitioned among the processors at the root of the tree. When a processor has searched the section of the original tree allocated to it, the processor becomes idle and is not given any additional work to do. The reassign jobs strategy extends automatic partitioning by giving work to idle processors. This extension allows automatic partitioning to begin from any node in the search space, not just from the root. Adaptive control includes any control methods where the strategy is altered by events which occur during the execution of the problem. All of the control strategies shown in Figure 7.1 are described in this chapter (Sections 7.5 - 7.9) with the exception of the adaptive control strategies.

Parallel backtracking control strategies [Clocksin and Alshawi 1988, Alshawi and Moran 1988] initiate independent depth-first searches on separate processors. These strategies have been motivated by sequential iterative depth-first deepening algorithms [4, Korf 1985] and parallel formulations of depth-first search[1]. In all of these parallel algorithms, the load of one processor is split among other processors either by sending environment information such as stacks, or recreating these stacks through the following of oracles. In the case of the Delphi research, oracles are used exclusively to partition the work load. In the backtracking control strategies described in this research, oracles are not used exclusively; it is the combination of tree partitioning algorithms implemented through the use of oracles which allows a search space to be partitioned. The backtracking strategies described in this research are not only related to the previous Delphi research (since oracles are used), but also to many of the tree partitioning strategies developed in the area of game playing[3]. In particular, the partitioning algorithms used by these backtracking strategies are very similar to the two tree splitting algorithms in papers by El-Dessouki and Huen [1980][5] and Ali [1987].

4 Slate, D.J. and Atkin, L.R., Chess 4.5–The Northwestern University chess program. In *Chess Skill in Man and Machine*, Springer-Verlag, New York, 1977.

5 El-Dessouki, O., and Huen, W., Distributed Enumeration on Between Computers. *IEEE Transactions on Computers*, Vol. C-29, No. 9, September, 1980.

The tree partitioning algorithm described in El-Dessouki and Huen [1980][5] is equivalent to the *automatic partitioning* algorithm described in Section 7.8 with minor differences in the splitting schemes. This algorithm is described in some detail to contrast it with *automatic partitioning* and the *reassigning jobs* strategies described in Sections 7.8 and 7.9 respectively.

Two types of enumeration methods (backtracking algorithms) are pointed out as candidates for parallel formulation in El-Dessouki and Huen [1980][5]: those which require all solutions to a problem and those which only have a single solution[6]. The parallel algorithm is described in terms of subprocesses which exist in one of the following phases:

(1) Selection phase.
(2) Full enumeration phase.
(3) Exchange phase.

Phase 1 – The selection phase begins at the root of the search space with the number of cooperating subprocesses (NCP) know by each separate subprocess. The nodes are distributed starting from the root node with the final result being that the tree is automatically partitioned so that each subprocess explores an independent subtree. The splitting algorithm is shown below where SIZE denotes the number of nodes which can be distributed. The subprocess numbered MYNUMR is assigned node *g* where:

$$g = [(\text{MYNUMR} - 1)\ \textbf{modulo}\ \text{SIZE}] + 1$$

New values are then calculated for MYNUMR and NCP.

$$\begin{array}{ll} \text{DELTA} = 1 & \text{if } g \leq [(\text{NCP} - 1)\ \textbf{modulo}\ \text{SIZE}] + 1 \\ \text{DELTA} = 0 & \text{Otherwise} \end{array}$$

$$\text{MYNUMR} = [\text{MYNUMR}/\text{SIZE}]$$
$$\text{NCP} = [\text{NCP}/\text{SIZE}] + \text{DELTA}$$

When SIZE becomes equal to or greater than NCP, each process picks the number of nodes equal to [SIZE/NCP]. If SIZE is not divisible by NCP, the last process takes any remaining nodes. The selection phase is then ended and full enumeration begins.

Phase 2 – Full Enumeration is the phase where each process explores its own unique subtree. Since the parallel algorithm described by El-Dessouki and Huen is intended mainly as a distributed branch and bound technique, additional tasks are created for the sending and receiving of information (such as the upper bounds) between processes. For the purpose of this comparison, it is sufficient to say that this phase performs a standard depth-first search of its subtree. When the exploration of the subtree is completed, the process enters the exchange phase.

6 There is no connection between the enumeration methods described in El-Dessouki and Huen [1980][5] and any integer labelling of a search space.

Phase 3 – A process which is idle (process *p*) obtains additional work to do by interrupting a neighbouring process (process *q*) and requesting a portion of *q*'s search tree. Each process is tried in turn until one of the other processes is able to split off some of its load. If none of the other processes are willing to give up a portion of their search space, process *p* informs the group that it is quitting and halts. An interrupted process (process *q*) is willing to split up its search space if the variable UNSEARCH points to an unsearched node. When the enumeration phase begins, UNSEARCH is initialised to the highest right-hand node in the search space. If the process has been given more than one node to explore (i.e., [SIZE/NCP] > 1) UNSEARCH points to the rightmost node received. If there is only one node to explore, then UNSEARCH is the rightmost son of this node. There are additional rules which describe whether an exchange is performed when a process interrupts and requests one.

If the exchange is successful, process *p* receives the address in *q*'s UNSEARCH and any information needed to switch its search to this new subtree. Full enumeration is again entered to search this new subtree possibly resulting in additional exchanges being made between processes. The distributed algorithm continues with each process entering full enumeration and the exchange phase until the original tree has been searched exhaustively. An important point to notice here is that the original phase involving selection is only performed at the root of the search space. This selection phase is never entered again once full enumeration begins.

As a comparison with the Delphi strategies, the selection phase of this algorithm is equivalent to the automatic partitioning strategy of the Delphi machine. Automatic partitioning is used as the initial phase of all backtracking control strategies, so the selection phase of the El-Dessouki and Huen algorithm is the same as the initial phase of all backtracking control strategies. Likewise, the full enumeration phase can be considered equivalent (though El-Dessouki and Huen include many extra parameters to allow branch and bound searching) to the standard depth-first search performed by the Delphi Path Processors. It is the third phase which involves the exchange of information between processors which is different in the two systems.

El-Dessouki and Huen describe a method of exchanging information which involves two processors. One processor interrupts the other, and then state information is transferred to split up the work. The work is split so that the receiving processor gets one branch of the sending processor's search space. The Delphi backtracking algorithms described in this chapter have the following additional features:

- More than one processor can be included in an exchange.
- Any state information sent is in the form of an oracle.
- The application of automatic partitioning starting from any node in the search space (as described by an oracle) is the method of splitting up the work load.

This final statement is the crux of the backtracking control strategies developed in this research. This is also what makes them unique among other tree partitioning algorithms. In the El-Dessouki and Huen algorithm the automatic partitioning only occurs at the root of the search space. Any partitionings which occur after the root has been searched, are performed by a separate algorithm. In the Delphi backtracking strategies, all partitionings of the search space are governed by the same algorithm.

Oracles allow the automatic partitioning algorithm to be applied uniformly to any node in the search tree. Following an oracle recreates the environment at a particular node in the search tree, and this node can be considered the root for an additional application of automatic partitioning.

The Ali algorithm [Ali 1987] is an extension of El-Dessouki and Huen which adds the ability to partition starting from any node in the search space. This is performed by copying all or part of the environment of an interrupted processor to the idle processors, and then adjusting that environment (in some cases by backtracking) before partitioning is initiated. The initial splitting scheme used by Ali has been described in Chapter 2, Section 2.4.5. The initial splitting algorithms of all three models [El-Dessouki and Huen, Ali and Delphi (see Section 7.8)] are equivalent. The method of exchanging information by the use of oracles is unique to Delphi.

Other models involve the use of oracles for the recreation of the environment at particular nodes in the search space. Chapter 2, Section 2.5 describes some related work involving the use of oracles. Both of the previous papers on Delphi machines [Clocksin and Alshawi 1988, Alshawi and Moran 1988] involve the use of oracles and backtracking control strategies, but none of these strategies employs any tree partitioning algorithms. In these models, the generation of oracles is the method for dividing up the search space among multiple processors. The same is true for the non-backtracking control strategies described in this chapter; it is the generation and following of oracles which is the method for partitioning the search space. However, in the backtracking algorithms described in this chapter, it is the application of automatic partitioning which is the method for dividing the search space; the use of oracles provides the means.

There is one other feature that the algorithms described in this chapter have that others [Clocksin 1987, Clocksin and Alshawi 1988, El-Dessouki and Huen 1980, Ali 1987] do not—all of the non-backtracking control strategies have been implemented and most of the backtracking control strategies have been both implemented and tested (see Chapter 8). Only one other implementation of the Delphi machine is known [Alshawi and Moran 1988] and the control strategies investigated in that system are more similar to the non-backtracking control strategies developed in the current research. As for the backtracking control strategies, there are parallel implementations of backtracking algorithms [for example, Finkel and Manber 1987], but none are known to split a search space by automatic tree partitioning.

In summary, the use of oracles to recreate an environment is a unique feature of parallel Prolog models based on the original Delphi machine [Clocksin and Alshawi 1988]. Other parallel Prolog models [Ali 1987 and models discussed in Chapter 2, Section 2.4] use some method of environment sharing or copying to perform the same task as that of following an oracle. In addition to the use of oracles, the backtracking algorithms described in this chapter use tree partitioning strategies as the method for dividing up a search space. Other parallel Prolog models which use oracles [Clocksin and Alshawi 1988, Alshawi and Moran 1988, Shapiro 1989] do not use partitioning strategies to divide the search space. It is the generation and following of oracles that partition the search space.

Tree search algorithms [El-Dessouki and Huen 1980, Ali 1987] may partition the search space in a manner similar to the backtracking control strategies developed in this research, but they do not use oracles for the recreation of an environment. It is the combination of these two features which makes the backtracking algorithms in this chapter uniform, efficient (see Chapter 9) and unique.

7.2 Non-Backtracking Strategies

With a non-backtracking strategy no choice points (such as those created by sequential Prolog systems) are built. Since there is no choice point stack (this term will be used for a contrived data structure containing only choice points) maintained locally, the search can only progress downwards from the root of the tree structure. When a branching point is reached there is no alternative but to save the current position to later be explored by the same or a different processor. Saving the current position is analogous to creating a choice point but instead of the choice point information necessarily being kept by a local process (one of the PPs), it can be kept 'globally' by the Controller process. This leads to the basic design of the non-backtracking strategies; the Controller is used as a global (available to all of the Prologs) choice point stack. The interesting feature is that the data kept on this stack is an oracle giving a complete description of how to proceed to a particular place within the search space. It does not contain conventional choice points which consist of pointers into other local stacks. The information held by an oracle is truly global and can therefore be given to any of the Path Processors.

The term 'Path Processor' is used throughout this document to indicate the running of one Delphi Prolog server or equivalently, Prolog system. A Path Processor is a piece of software and not a specific piece of hardware. Though the Delphi machine is most often configured to run exactly one Path Processor per μVAX, it is possible to run numerous Path Processors on any of the host machines.

7.3 Conventional Search using Oracles

We consider just how a non-backtracking single Delphi processor might operate using oracles instead of a conventional choice point stack. The well known search strategies breadth-first, depth-first iterative-deepening, and depth-first are described using oracles to control the search. The following program and its representation as a tree (shown in Figure 7.2) is used in the example searches.

```
root      :-   branchl.
root      :-   branchr.
branchl   :-   c1bl.
branchl   :-   c2bl.
branchr   :-   c1br.
branchr   :-   c2br.
branchr   :-   c3br.
```

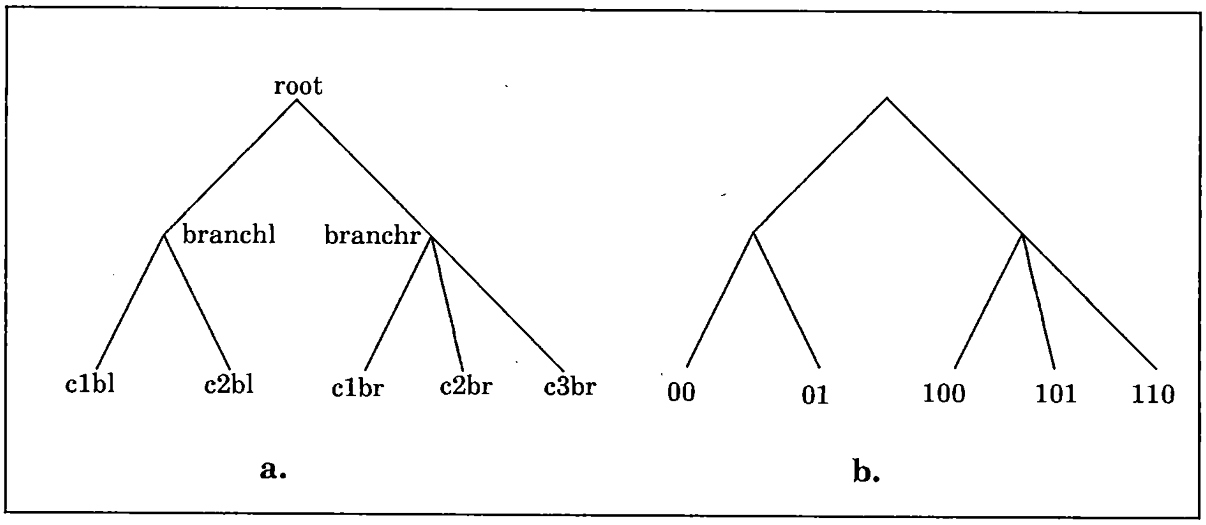


Figure 7.2 Example Tree

Figure 7.2a is a representation of the Prolog program using the goals of each clause as the nodes of the tree. The representation of Figure 7.2b shows the oracles required to reach each of the leaf nodes in the program. Naïve versions of breadth-first and depth-first iterative-deepening search [Korf 1985] can be easily demonstrated using oracles instead of a choice point stack to control the search. These algorithms generate and explore oracles without knowledge of what the search space looks like at any particular point. Many nodes are redundantly explored and many of the generated oracles are superfluous. For this reason these algorithms are termed naïve. For each of these naïve searches the oracle is generated and explored starting from the root of the tree.

The following algorithm uses oracles to perform a breadth-first search:

```
N = 0
DO {
    N = N + 1
    generate all oracles of length N
    explore all of these oracles
} UNTIL there are no incomplete oracles of length N
```

Depth-first iterative deepening using oracles can be implemented by the following algorithm:

```
N = 0
DO {
    N = N + 1
    generate all oracles of length N
    explore each of these oracles as they are generated
} UNTIL there are no incomplete oracles of length N
```

An incomplete oracle is an oracle which is not long enough to reach a leaf node. When an oracle is complete it reaches a leaf node or may extend beyond it. For example, the oracles [01] and [01001] are both complete. Both of these oracles reach the second leaf node (counting from the left) of the tree in Figure 7.2.

Storage is the main difference between the naïve breadth-first and depth-first iterative-deepening algorithms. In the breadth-first algorithm all paths of length N are generated and stored. Each of these oracles is then in turn explored. In the second algorithm (depth-first iterative-deepening) only one path is generated at a time. As soon as the path has been generated it is explored and the oracle is discarded. Depth-first iterative-deepening shows a time complexity similar to a breadth-first algorithm, but has the space complexity of a conventional depth-first algorithm. A more detailed examination of depth-first iterative-deepening search can be found in Korf [1985].

In both the naïve breadth-first and depth-first iterative-deepening algorithms oracles are explored in the following order:

[0], [1], [00], [01], [10], [11], [000], [001], [010], [011], [100], [101], [110], [111]

The difference between sequential Prolog search (which is depth-first) and the naïve searches using oracles for control is in the amount of information available and used at each branching point. For the sequential Prolog what is known at a branching point is that there is at least one more branch that needs to be explored. A pointer (the choice point) is set up to allow the future search of the next right-hand branch and ultimately all others. The naïve searches make no use of any information available when a branching point is reached; even the fact that a branching point has been reached is not used. They function by generating and exploring all oracles of length one up to the maximum depth of the tree. The information available during the search is that an oracle is incomplete or complete. Depth-first search using oracles can also be demonstrated using only this information:

```
Oracle = [0]
CALL Extend-With-Zeros
DO {
    remove all trailing '1' bits from Oracle
    flip last bit in Oracle to '1'
    CALL Extend-With-Zeros
} UNTIL there are no '0' bits in Oracle

SUBROUTINE Extend-With-Zeros
explore Oracle
WHILE Oracle is incomplete {
    concatenate a '0' bit onto the end of Oracle
    explore Oracle
}
```

Using this naïve depth-first algorithm, oracles are explored in the following order:

[0], [00], [01], [1], [10], [100], [101], [11], [110], [111]

With the *WAMO* (WAM with Oracle control) instructions an extra piece of information is available at each branching point; the exact number of branches which need to be explored is known before a particular branch is chosen to explore. This information is given by the single argument of the `setmax` instruction. If the current branch is the only one that needs expanding (deterministic choice) this argument is 1. If there are additional branches this argument is greater than 1. The naïve breadth-first, depth-first iterative-deepening and depth-first algorithms do not make use of the information provided by the `setmax` instruction or even that a branching point has been reached. If the argument of `setmax` is used during the search more efficient algorithms can be designed. An example of a depth-first search which uses the number of choices at a branching point is shown in Figure 7.3.

Figure 7.3a shows the start of the process with the stack of jobs (oracles to be explored) containing an entry for root. At the root of the tree the `setmax` instruction would have an argument of 2 showing that there are two clauses to choose from at this node. The two oracles describing these two choices go on to the stack pushing the rightmost branch first through to the leftmost. Figure 7.3b shows the condition of the stack after these two oracles have been pushed onto it. The dark arrows represent the furthest point in the tree (from the root) that is reached for each oracle explored.

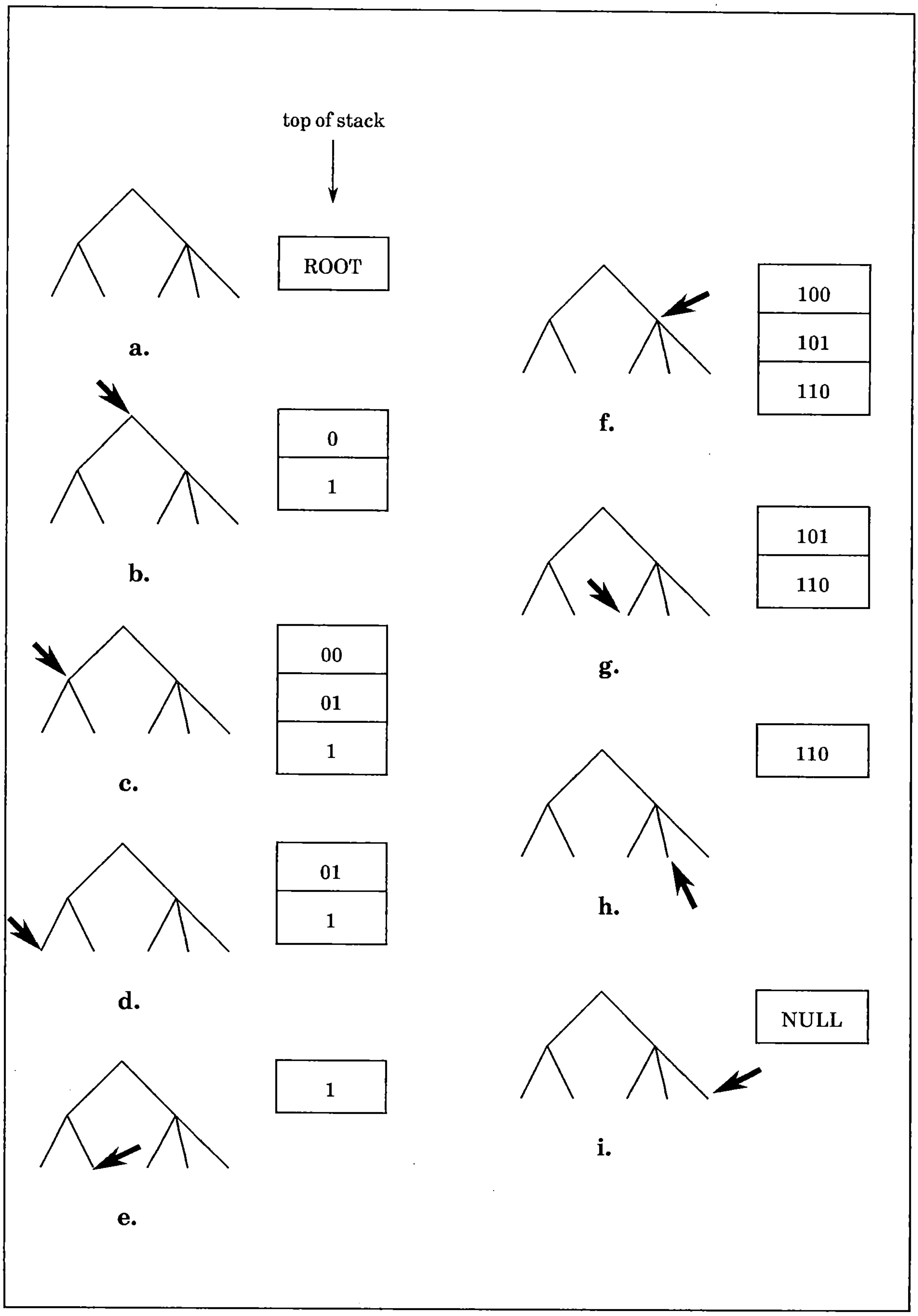


Figure 7.3 Using Oracles and Setmax Information for Depth-First Search

So far no oracles have been explored so the arrow is shown at the root of the search space. In Figure 7.3c, the top oracle [0] is popped and explored. A branching point is reached adding two new jobs onto the stack. The process continues in this manner; popping the stack to obtain the next oracle to explore and pushing new entries when a branching point is encountered. When there are no more jobs available on the stack (Figure 7.3i) the tree has been searched. The important points to notice from this example are:

- Each oracle or job starts from the root of the tree.
- At each branching point, an entry (or entries) has to be put on the stack or information on the existence of alternate choices will be lost.

The searches described in Section 7.5 fall into three categories depending on the amount of information used at each branching point:

(1) No information about the branching point is used.
  - naïve breadth-first
  - naïve depth-first iterative-deepening
  - naïve depth-first

(2) The fact that a branching point has been reached is used.
  - sequential Prolog depth-first search
  - depth-first search shown in Figure 7.3

(3) The number of branches is used.
  - depth-first search shown in Figure 7.3

In the naïve depth-first search the oracles [10] and [11] had to be generated and explored. With the depth-first search using the `setmax` information (Figure 7.3) these two oracles never appeared on the job stack. The reason is that the `setmax` instruction not only indicates the number of available choices, but also provides the encodings for each of these choices. When the node with three branches was reached (Figure 7.3f), the oracles pushed onto the stack were those defined by Figure 7.2b. The method for encoding the oracles shown in Figure 7.2b is described in Section 7.4.

7.4 Encodings

When a tree with a maximum branching factor of two is encoded, all left hand branches are the '0' choice, and all right hand branches are the '1' choice. If there is only one branch it can arbitrarily be called the 0 or 1 choice; we have used 0 to represent a single branch. When the number of branches is not a power of two the branches can be encoded in various ways. Two encoding methods named *regular* and *compressed* are demonstrated in Figure 7.4. The example tree is the same as the one used in Figure 7.2

Compressed encodings uniquely enumerate the clauses of a predicate using a smaller number of bits overall than the regular encoding. In Figure 7.4 we are interested in encoding the three alternative clauses for `branchr` (see Figure 7.2a). There are three clauses or objects to be encoded.

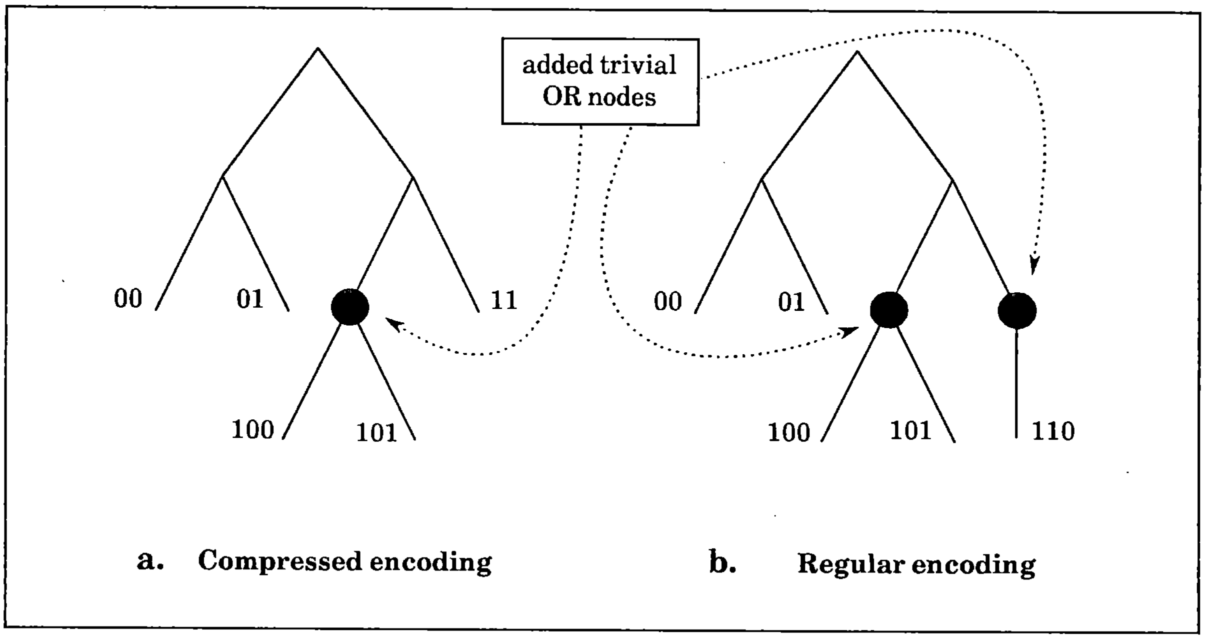


Figure 7.4 Two Encoding Methods

The compressed encoding (Figure 7.4a) shows two clauses encoded with three bits, and one clause with two bits. If we consider these three branches on their own, the number of unique bits is two for two of the clauses and one for the final clause (the initial '1' bit is similar for all three leaf nodes).

With three objects a compressed encoding uses two bits for two of the objects and one bit for the third object. The combined number of bits needed for encoding all three objects is five. With the regular encoding, three items are encoded with two bits each giving a total of six bits. In general, a regular encoding of n items entails $n\lceil \log_2 n \rceil$ bits for all encodings. A compressed encoding uses $n(\lceil \log_2 n \rceil + 1) - 2^{\lceil \log_2 n \rceil}$ bits. See Appendix 7a for a discussion of these figures. Reasons can be given for using either of these encodings. The regular encoding method was more often chosen when implementing Delphi control strategies for the following reasons:

- Space Conservation
  One reason to use compressed encodings is for extra space conservation. Packing the bits with regular encodings proves to be enough of a space saving and keeps the transmission times low.

- Representation
  In a regular encoding all clauses in a predicate have the same number of bits. This makes the decoding algorithm simple and if graphically displayed, all clauses are located at the same level of the tree.

- Complexity
  With a regular encoding the integer corresponding to the encoded string is equal to the clause number being picked. If the clause numbering starts from one then the encoded string is equal to one less than the clause number.

7.5 Expanding a Job

In the algorithms described so far, the storage for nodes to be expanded (oracles to be followed) has been assumed to be local to a single processor. The computation involved in exploring each of the oracles generated is also assumed local. Oracles are global data in that they give a complete description of the path to be followed from the root of the tree. With a Delphi machine these oracles can therefore be distributed to any available host processor. *Expand a job* is the first strategy which will demonstrate the distribution of oracles to more than one Path Processor.

In the naïve breadth-first strategy the only information used was that a leaf node had been reached by all generated oracles. Expanding a job additionally uses information when an internal node has been reached. The information consists of two messages NEEDNEWJOB and NEEDEXT which are sent by the Path Processors to the Controller. The need a new job message is sent on two occasions—when the processor is originally initialised and after a leaf node of the tree has been reached. The need an extension message is sent when an internal node which has a branching factor greater than one is reached. If a deterministic node is reached (branching factor equal to one), then the choice is automatically taken without any need for the Controller to send an extension of [0]. With these two messages a variety of *expand a job* strategies can be implemented.

Expanding a job in its most basic form is a simple breadth-first search using multiple processors. The Controller process maintains a queue of oracles in breadth-first order. It uses information returned by the Path Processors to avoid the creation of redundant oracles such as those generated in the naïve breadth-first strategy.

The naïve strategies generated the next oracle to explore by *guessing* the next bit in the oracle string. Guessing is when bit permutations of a certain length are created and explored without knowing that the oracle will be successful. For example, in the naïve breadth-first and depth-first iterative-deepening algorithms, all permutations of length n were created without considering whether a failed oracle of length n-1 would obviate generation of some oracles of length n. With the naïve depth-first algorithm an extension of one '0' bit or replacement of the '0' bit by a '1' was a guess that the newly generated oracle would be successful. All of the naïve algorithms guess ahead by a single bit. It is also possible to guess ahead by more than one bit. If the oracle [0110] ends on an internal node then we might guess that a successful oracle may be [0110000], three bits ahead of the known sure path.

The length of an oracle expansion which is not a guess is the number of bits needed to encode the choices at a branching point. If the tree is binary the expansion will be one bit. In general, with a branching factor of n the extension length can be up to $\lceil \log_2 n \rceil$ bits. The Controller is the process which generates oracles and their expansions. For the Controller to be ensured of correctly expanding oracles, information concerning the number of choices has to be communicated from the Path Processors back to the Controller. This information is the parameter of the setmax instruction which occurs at the beginning of a set of clauses. Without this information only guesses of one bit could be ensured of being correct expansions. The only information available at each node is the type of node that it is—internal or leaf node. With compressed encodings it is fairly obvious why one-bit extensions will always be correct. The Path Processors communicate to the Controller when they need an extension and an internal node has been reached. All internal nodes have exactly two outward branches so any one-bit extension will be correct. When a leaf node is reached no extensions are generated as the Path Processors then ask for a new job to be given to them by the Controller.

Deterministic choices can be made automatically without involving communications to the Controller. For this reason one-bit extensions are also correct when using regular encodings. Just as with the compressed encodings, communications to the Controller will only be performed if there are two branches to choose from or a leaf node has been reached. An example of a deterministic choice being automatically taken is shown in Figure 7.5. When the oracle [11] is sent to Path Processor 2 this processor automatically explores route [110].

With a *bit by bit* strategy oracles are expanded one bit at a time. The jobs queue is initialised to hold the two oracles [0] and [1]. As Path Processors (PPs) report to the Controller with the message NEEDNEWJOB, they are given the next oracle on the jobs queue. When PPs report back with the message NEEDEXT, a '0' bit and a '1' bit are appended to the end of the oracle which that PP has just followed. These two newly created oracles are placed at the end of the jobs queue. This strategy is termed *strictly following* meaning that the Path Processors have only one mode of operation; to follow the oracle that they have been given beginning from the root of the tree. An example of this bit by bit strategy operating with two host Path Processors is shown in Figure 7.5.

The jobs queue and last run array are two data structures manipulated by the Controller. The last run array holds the last oracle which was explored by each of the Path Processors (PPs). In Figure 7.5, PP1 and PP2 are the first and second locations respectively in the last run array (with location number one on the left). Processing is shown as if it occurs in a synchronous manner with the Controller distributing oracles and the Path Processors responding. Though synchronous execution does not really occur, it is shown in this form for the sake of simplicity. The Controller updates the queue and last run array as each PP sends back either of the two possible messages. Darkened areas on the trees to the right of the PPs show which path is explored after the PP's request is granted. As the processing is asynchronous, it is possible that a strict breadth-first search will not be performed. If we assume that all oracles of the same length take the same time to compute and that all Path Processors report back to the Controller with equal priority, then this control strategy will effect nearly breadth-first search. In addition to the problem of asynchronous behaviour, exact breadth-first search will not occur since deterministic branches are automatically explored.

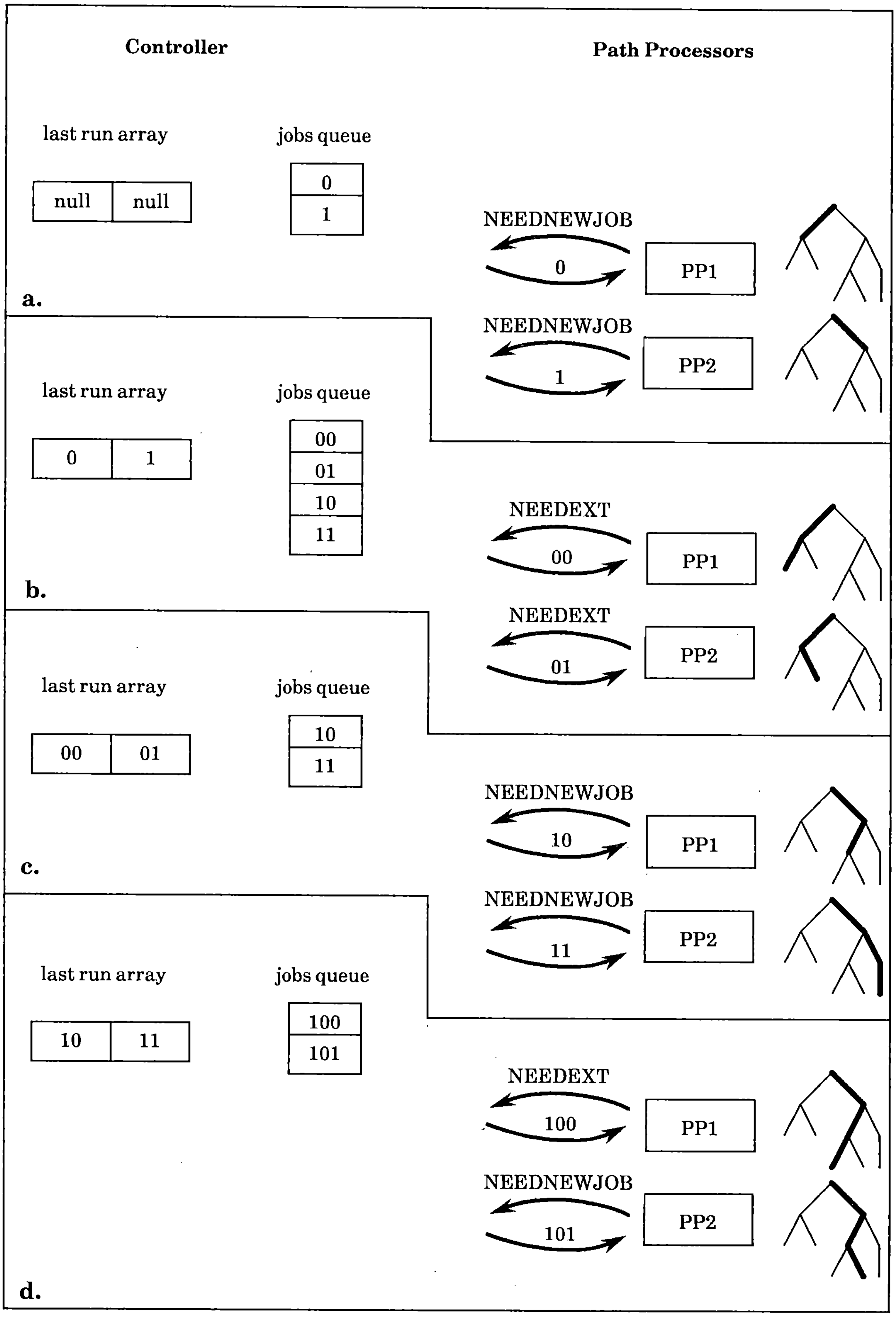


Figure 7.5 Bit by Bit Strategy

Figure 7.5a shows the first step in the bit by bit strategy. The jobs queue is initialised with the oracles [0] and [1]. The last run array is empty as no oracles have been distributed yet. Each of the two PPs sends a NEEDNEWJOB message to the Controller shown by the top curved arrows. Communications to the Controller from the Path Processors are shown by the top arrows with communications to the Path Processors in the bottom arrows. The Controller responds to these messages by sending oracle [0] and [1] to PP1 and PP2 respectively. The sending of these oracles from the Controller is shown in the bottom curved arrows. The last run array can now be updated showing the latest oracles which have been distributed (see Figure 7.5b). The paths explored by each of the PPs is shown by the trees to their right. Both Path Processors reach a choice point and so each will send a NEEDEXT message.

In Figure 7.5b NEEDEXT has been sent by both PPs. The jobs queue is updated to reflect the new paths which have been found by the PPs. When PP1 sends its NEEDEXT message the oracle in the first position of the last run array is extended by one bit. The two new resulting oracles [00] and [01], are placed on the jobs queue. The same is true for PP2. When its NEEDEXT message is received [10] and [11] are placed on the queue. The first two jobs on the queue are sent by the Controller to the PPs to explore. In general, the jobs queue expands by one oracle for each NEEDEXT message received. A NEEDEXT sent by a PP creates two new oracles on the jobs queue but also takes the first oracle in the queue as that Path Processor's next job to explore. The net effect of a NEEDEXT message is to increase the queue by one job while a NEEDNEWJOB message reduces the jobs queue by one oracle.

Figure 7.5c shows the last run array updated with the oracles last sent to PP1 and PP2. Both PPs reached a leaf node with their last exploration and so they each send a NEEDNEWJOB message. PP1 is given [10] to explore and PP2 is given [11]. PP1 reaches an internal node and sends back a NEEDEXT because of this (see Figure 7.5d). PP2 explores [11] which leads to a deterministic branch. This branch is automatically taken without any communication to the Controller. PP2 reaches a leaf node and has to request a new job.

In Figure 7.5d the NEEDEXT message of PP1 puts two jobs onto the queue. The first of these is immediately taken by PP1 to explore. PP2 gets the final job. The search is known to be complete when all Path Processors are blocking on a NEEDNEWJOB message and there are no more jobs in the jobs queue to give them.

Many variations on this job expansion theme can be created. For example, it is possible to send an extension which represents an additional number of bits to be appended onto the previous attempted path. This saves on having to always start the computation from the root of the search space. The next strategy described, branch by branch, uses a technique to extend a path and avoid always starting at the root of the search space.

7.6 Branch by Branch

Branch by branch is a strategy which demonstrates the limit of the bit by bit strategy. It avoids always having to start from the root of a tree when a choice point is reached by allowing the Path Processor to pick a branch. For this reason it is more efficient than the bit by bit strategy. In the bit by bit strategy the Controller communicated oracles to the Path Processors and the PPs responded with one of two messages. In this strategy oracles are communicated both by the Controller and by the PPs. Branch by branch is a non-backtracking strategy which exhibits features of both a depth-first search and a breadth-first search.

Since a non-backtracking strategy is being used, the minimal number of communications that can be involved is equal to the number of choice points in the tree, a choice point being where more than one alternate clause is available to choose from. We are assuming here that no local stack is maintained to store some of the potential oracles and that there is no omniscient routine for guessing a correct path. At each choice point the PP must report that there is more than one branch available as it is unable to backtrack and try these branches by itself. It can however choose one of the branches to try while reporting one or more of the other alternatives. If this technique is used the result will be to keep each Path Processor exploring a particular branch from the root of the tree to a leaf node. Only when a leaf node is reached does the PP report back to the Controller and request a new oracle to explore.

For the branch by branch strategy the NEEDEXT message is no longer needed. An alternate message, GIVEPATHS, is used in its place. When a choice point is reached this message is sent along with the oracles for paths which the PP cannot explore itself. In one possible implementation, these oracles are paths from the root of the tree to the current node with extensions for all branches excepting the clause that the PP picks to explore itself. Figure 7.6 shows a particular branch by branch strategy operating. In this case, the clause automatically chosen by the PP is always the first clause or leftmost branch.

Computation proceeds down the pages of Figure 7.6 with the Controller's actions shown on the left-hand side and the two host PPs shown on the right. The jobs queue is initialised to hold the first job which is the root node. The processor which receives this job will know that it is supposed to immediately pick the first clause at the first choice point and continue to pick the first clause at each choice point until a leaf node is reached. As each choice point is encountered, a message (GIVEPATHS) including the paths of all oracles which are not explored by this processor is sent to the Controller. The return message after a GIVEPATHS is performed is an acknowledgment (ACK) that the paths have been received. If a new job is requested and there are no jobs in the queue, the processor must wait on the idle processor stack (not shown) until a job has been enqueued by another processor doing a GIVEPATHS.

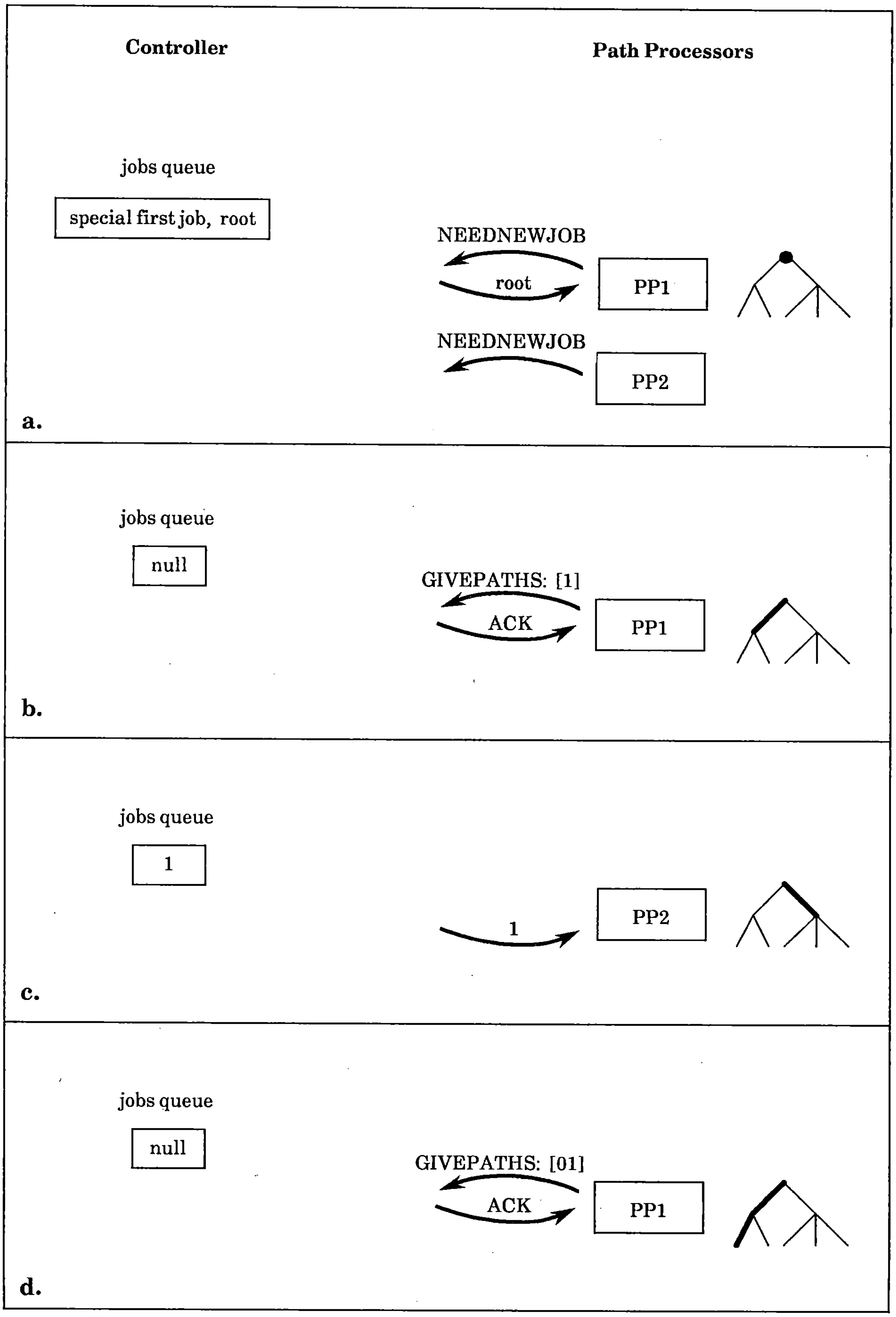


Figure 7.6 Branch by Branch Strategy

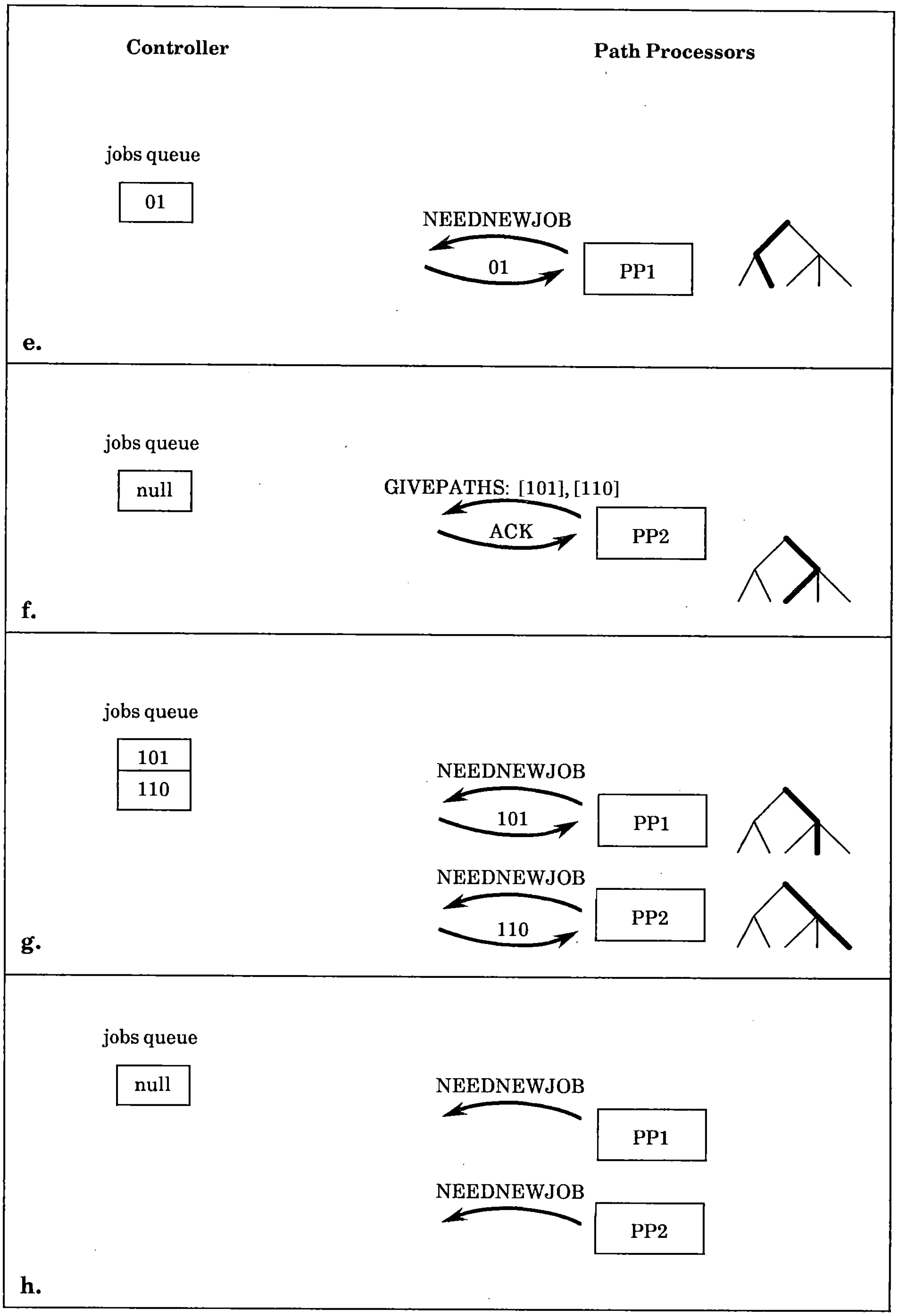


Figure 7.6 Branch by Branch Strategy (continued)

Two distinct modes of operation are encountered within this strategy: a following mode and an automatic choice mode. The following mode is identical to the following performed in the bit by bit strategy. An oracle is sent to the Path Processor which follows that oracle as far as it leads. After the following mode is completed an automatic mode is entered involving automatically picking a particular numbered clause. This mode is continued until a leaf node is reached when the Path Processor has to request a new job. This automatic picking mode is also accompanied by the transmission of untaken paths back to the Controller as each choice point is reached.

Figure 7.6a shows the start of the branch by branch strategy with the jobs queue initialised with the root node. Both PPs are requesting a new job after their initialisation phase. There is only one job available and it is given to PP1. PP2 must wait until a new job is available from the jobs queue. At the root PP1 sees that there are two branches. Since it cannot backtrack PP1 is only able to explore one of those paths. It transmits the untaken path [1] to the Controller and takes the [0] path (Figure 7.6b). In Figure 7.6c the path transmitted by PP1 is immediately given to PP2 which has been blocked waiting for a job to appear on the queue. PP1 reaches a choice point after automatically taking path [0]. At this choice point PP1 sends the path it will not explore [01] to the Controller with a GIVEPATHS message (Figure 7.6d). It then automatically proceeds to location [00] picking the first clause in the choice point to explore itself. It did not restart computation from the root of the search space to get to [00], it just automatically picked the first clauses of each choice point after its initial oracle (in this case the root node) was received.

Figure 7.6e shows PP1 requesting a new job and being given path [01]. PP2 reaches the choice point with three branches located at [1]. Two of these paths are transmitted to the Controller [101] and [110] after which the PP automatically proceeds to location [100]. In Figure 7.6f both PPs are requesting new jobs. After following the oracles they are given both PPs reach leaf nodes and again must request new jobs. When all participating PPs are blocked waiting on new jobs and the jobs queue is empty, the search is complete (Figure 7.6h).

One possible variation on the branch by branch strategy could be to pick the rightmost branch instead of the leftmost. Another branch by branch variation could be implemented to reduce communications over the network. Instead of communicating paths at each branching point, the Path Processor could store all of the oracles it encountered and transmit them all together. Figure 7.7 demonstrates this variation. The Path Processors do not perform a GIVEPATHS until they reach a leaf node. This can be seen in Figure 7.7b where PP1 does not do a GIVEPATHS until the leaf node is reached.

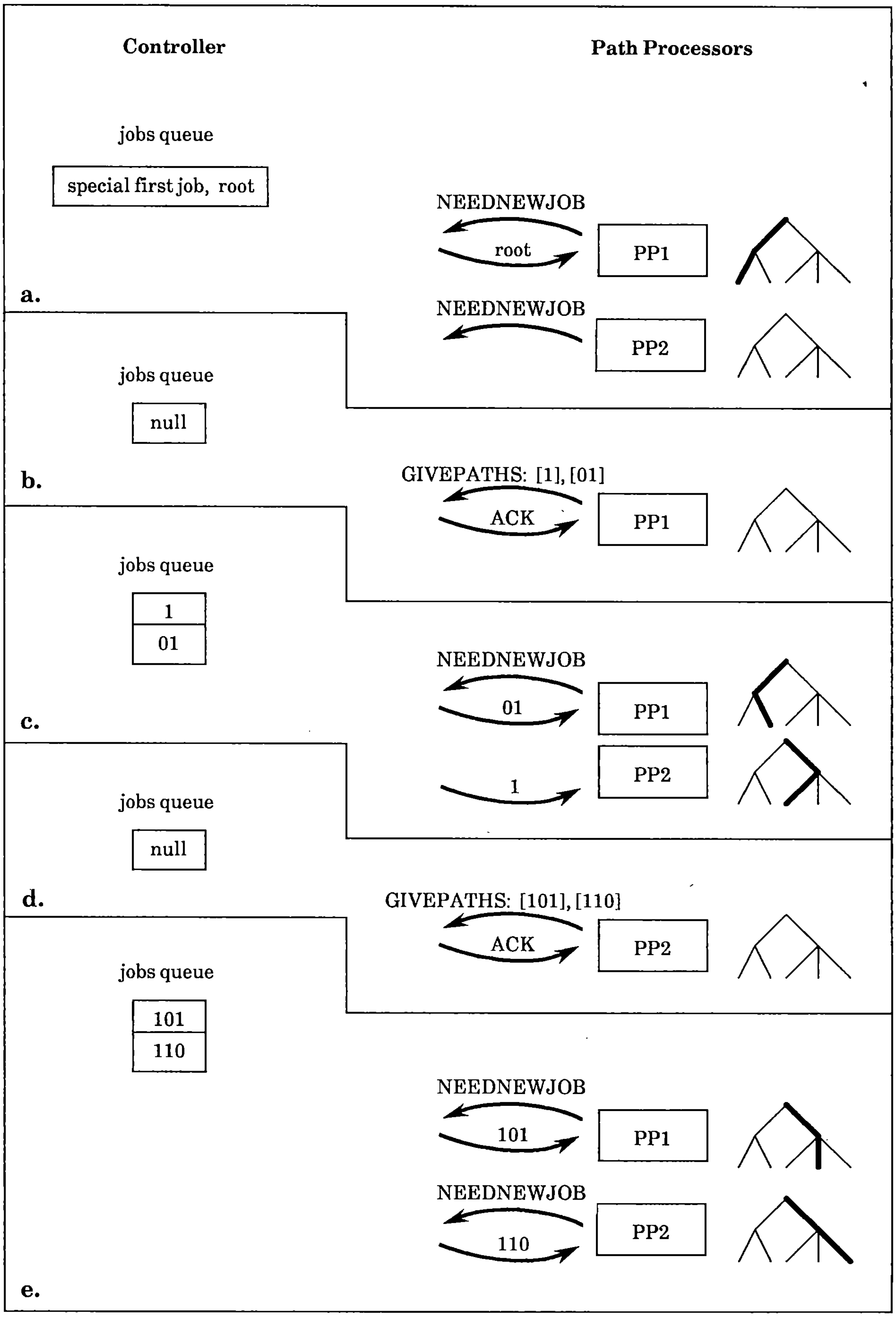


Figure 7.7 Branch by Branch Variation

Branch by branch can be considered a combination of depth-first and breadth-first search as it contains characteristics of both. When a processor reaches a choice point it reports all paths to the right of the one it explores. These paths are therefore generated in a breadth-first manner. When a Path Processor is involved in automatic picking, it proceeds in a depth-first manner always ending at a leaf node of the search space. Throughout this mode it gives paths back to the Controller in the order of a depth-first search as it explores further down the tree. Most of the backtracking strategies also have components of both breadth-first and depth-first exhaustive search.

7.7 Backtracking Strategies

Non-backtracking strategies have the major disadvantage that at every choice point, some operation to save untried branches must occur. There are two general ways to approach saving the untried branches:

(1) The extended oracle for each untried branch is placed on a local queue to await execution by the processor which discovered the path.

(2) Communications take place to the Controller, and the oracles representing untried branches are placed on the global jobs queue.

Combinations of the two choices above can also be constructed. The first method is analogous to the way sequential Prolog creates a choice point. The choice point stack of sequential Prolog becomes a local oracle queue. Oracles take the place of the pointers which need to be saved to recreate the environment before each choice is taken. The major difference is that the oracles recreate the environment starting from the root of the search space. They do not take advantage of any possibilities for replacing a portion of the environment without starting from scratch each time.

The second method indicates the communication of each untried path to the Controller. This entails the disadvantages of always having to explore each path starting from the root of the tree in addition to the communications overheads involved with sending and receiving oracles over the network. A compromise situation is to allow normal backtracking (as in sequential Prolog) to occur within particular sections of the search space. The search space is logically divided into separate sections to be explored by separate Path Processors. Normal Prolog control is applied to each of these sections with the Path Processors backtracking within their given portion of the original search space.

Partitioning the search space can be seen graphically as drawing separation lines through internal nodes of the tree. The branches within each section would all be explored by an individual backtracking Path Processor. Figure 7.8 shows examples of partitioning a search space by using horizontal or vertical lines to create the sections. In addition to using exclusively horizontal or vertical lines, a combination partitioning could be drawn using them together. It is very easy to draw the individual partitions. To effectively use the idea of partitioning, new control strategies have to be implemented which create and exploit sections of the search space, and properly map these sections onto separate Path Processors.

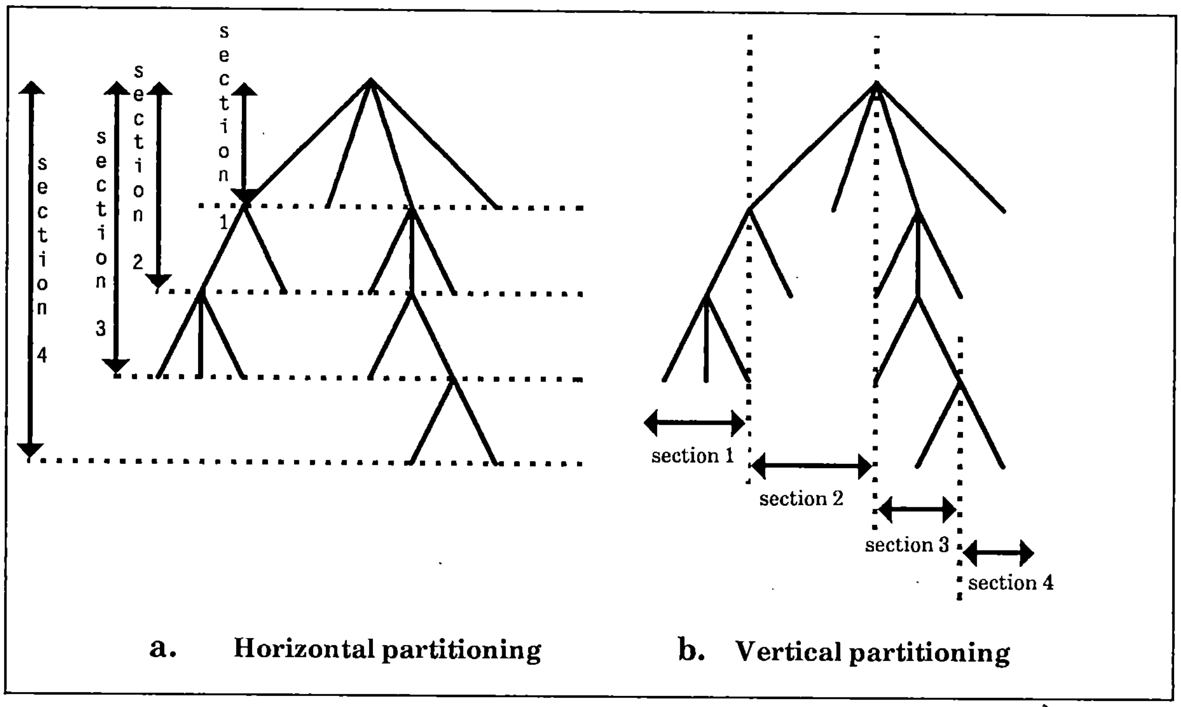


Figure 7.8 Partitionings

A horizontal partitioning leads to depth-first iterative-deepening strategies while vertical partitioning lends itself to separate depth-first searches. In Figure 7.8, the example tree is partitioned into four horizontal and four vertical sections. The four horizontal sections correspond to the four levels of the search space. For the vertical sectioning, four is just an arbitrary number. A vertical partitioning with one section would incorporate the entire tree while a partitioning with eleven sections would separate each of the eleven unique branches from the root of the search space to each of the eleven leaf nodes. The purpose of this partitioning is to divide the search space into sections which can then be searched by individual Path Processors with the intention of reducing the execution time of the Prolog program.

7.8 Automatic Partitioning

Automatic partitioning is a vertical partitioning strategy executed by each of the Path Processors without the need to communicate. Each of the Path Processors begins execution at the root of the tree. The search space is repeatedly partitioned until each Path Processor finds a unique section of the search space to explore.

Executing this strategy centres around the manipulation of two variables; the Unique processor number (U) and the number of processors in a Group (G). Both of these variables are initialised by the normal setup routines that the Delphi machine performs before execution of the Prolog program. The U value is initialised to the unique Global Server Number which each of the Prolog processes receives from the Controller. The G value is initialised to the number of Prologs initially configured for the Delphi run. During the computation G always represents the number of processors which are simultaneously executing this same portion of the search space. Variable U is each Path Processor's unique number which has a range from one to the value of G. Path Processors split up at choice points according to different partitioning algorithms. They continue with this splitting process until their G variable is equal to one. At this point there is only one processor available to explore all of the search space contained within the partition. This partition includes one or more branches from the choice point where the splitting occurred plus all of the tree located below these branches.

Partitioning is accomplished at a branching point with the creation of a *limited choice point*. The creation of a limited choice point enables a Path Processor to explore a subset of the available branches and not necessarily all of them. When a sequential Prolog choice point is created all of the branches will be explored when the same process backtracks to them. A pointer to the next clause to be tried is placed on the choice point stack. By storing this information and updating it upon backtracking, all clauses will eventually be tried in sequence. After the final clause is attempted the choice point is not needed any more and is released. This process relies on WAM instructions similar to `try`, `retry` and `trust` to perform the choice point creation, update and deletion functions. For a limited choice point to be manipulated the analogous `onum` instructions `onumtry`, `onumretry` and `onumtrust` are used. The information manipulated by the creation, update and deletion of a limited choice point is stored in a structure called the *oracle structure*.

Throughout the execution of automatic partitioning or any control strategy the path through the search space is maintained by the Path Processor in a data structure called the *current path.* If the oracle held in current path were followed it would lead to the location of that PP within the search space and recreate the environment associated with that location. Automatic partitioning itself may not need to store this information as it is never communicated to any other process, but previous non-backtracking strategies needed it and other backtracking strategies require the current path to be maintained. One reason for maintaining the current path, even with automatic partitioning only, is for debugging.

For non-backtracking strategies it is sufficient to use only the one `onum` instructing `onumsing`. When this instruction is executed a bit or bits are appended onto the end of the oracle contained in current path. As the current path is always extended with non-backtracking strategies, no previously appended bits are ever changed and the task is rather simple. With backtracking strategies the current path can change by adding, deleting, or updating any number of bits. To perform these tasks the oracle structure is used to hold information needed to maintain the current path so that it always reflects the location of that Path Processor within the search space. In addition to this function, the oracle structure maintains information on whether or not this branching point is being split with another Path Processor. If the branching point is being split with other PPs then a limited choice point is created.

The *current path* is a string of bytes containing an oracle. Associated with the string are byte and bit pointers which uniquely describe the next location for a bit or bits to be appended to the oracle. When a call to a predicate is made the values of these pointers are placed in the oracle structure along with the number of bits which are needed to uniquely pick a clause. Together, this information is enough to update the current path. A clause is chosen for execution, and the current path string is extended to reflect the branch that is about to be taken. If the clause fails and a redo is performed, the original bit pointer (held within the oracle structure) is reinstated as the current path's bit pointer. This sets the condition of the current path properly back to its state before the first clause was explored. The new bits to be appended will overwrite and update the previous ones. Figure 7.9 shows the two data structures needed to maintain the current path and use limited choice points.

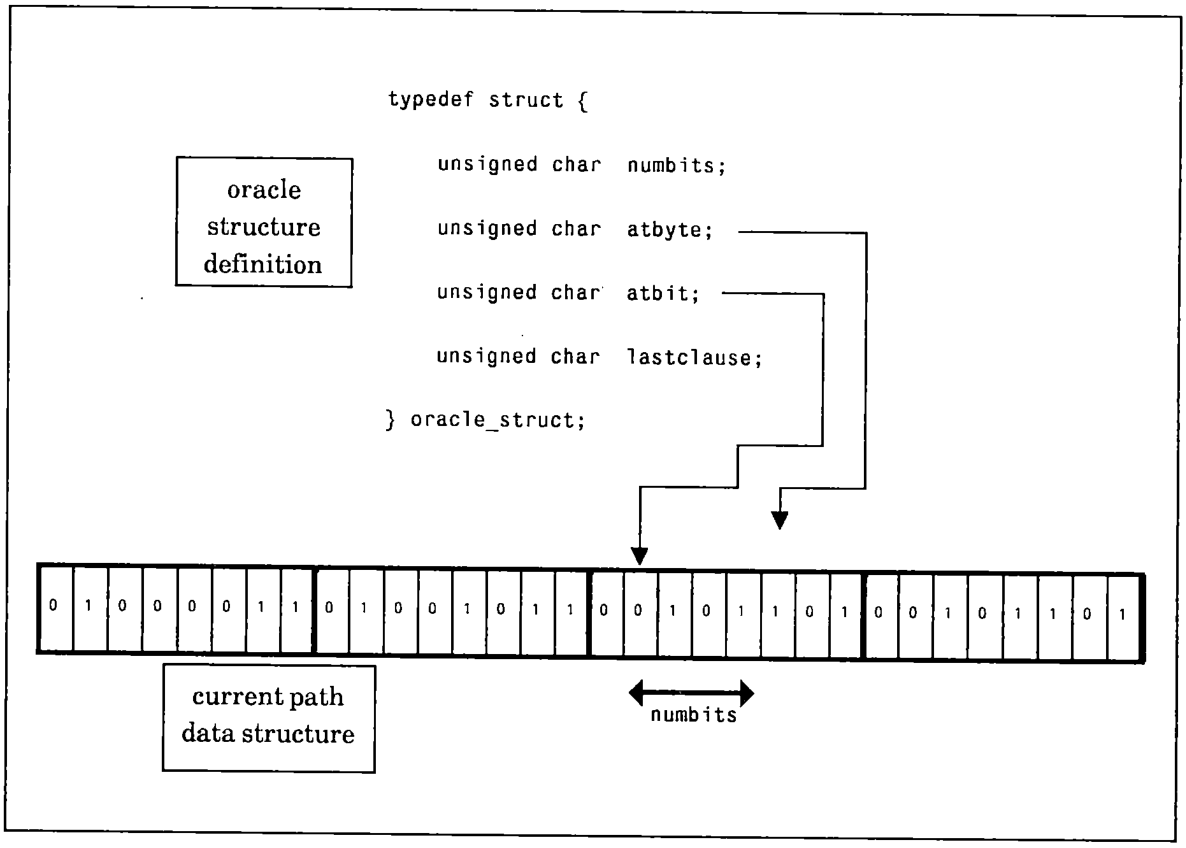


Figure 7.9 Oracle Structure

The oracle structure holds information needed to both create a limited choice point and to maintain the current path data structure. To create a limited choice point, the last clause which a PP has been assigned to explore must be saved during backtracking. Information must be retained throughout backtracking to properly update the path which describes that PP's location from the root of the search space. This information consists of two pointers and a small integer value. The pointers `atbyte` and `atbit` describe a unique bit location within the current path. The location specified by these two pointers is the position at which bits will be appended when a clause is chosen. When backtracking occurs the original values of these pointers will be returned. Character `numbits` contains the number of bits needed to encode the number of the clause which has been chosen next to explore.

In Figure 7.9 `atbyte` points to the third byte in current path and `atbit` the second bit within this byte. The number of bits needed to encode the clause chosen (`numbits`) to explore is shown as four. A small amount of information concerning the value of `lastclause` can be predicted from knowledge of the other three values. Assuming a regular encoding, the number encoded in the four bit positions starting at the location described by `atbyte` and `atbit` is five. The value of `lastclause` will be a minimum of five since the clause with this value has already been chosen to explore. The maximum value for `lastclause` will be fifteen since this is the largest number which can be stored in the four bits specified by `numbits`.

Within the oracle structure, `lastclause` is the number describing the last clause which is to be attempted by the Path Processor. If the choice point is normal and all clauses are to be tried as in a regular sequential Prolog, this number is set to the value of the `setmax` argument. Normal choice points are created when a single Path Processor is executing in solo (G is equal to one). If more than one Path Processor arrives at the same choice point (G is greater than one), then a limited choice point is created and the total number of choices are split among the group of participating Path Processors. Splitting algorithms can range from very complex methods of attempting to ensure that each Path Processor is given an equal share of the work load, to a simple mapping of branches onto unique identifier numbers (U's). A simple splitting strategy which uses such a mapping is demonstrated. It is called the *partition right* splitting algorithm and is a very right biased splitting strategy. Partition right assumes that the longer paths of the tree always exist beneath the rightmost branch. This strategy therefore assigns the largest possible number of Path Processors to this final branch.

Four cases exist for each splitting strategy relating the number of Path Processors available at the choice point to the number of branches. The four cases are:

- There is only one PP operating at this choice point.
- The number of PPs is less than the number of branches.
- The number of PPs is equal to the number of branches.
- The number of PPs is greater than the number of branches.

The first case is the most important. If there is only one Path Processor available to explore all of the branches of a choice point, then there is no need to execute a splitting algorithm at all. With this case the splitting algorithm code is never entered and the Path Processor behaves in a manner equivalent to sequential Prolog. If there are more branches than Path Processors, then the number one Path Processor in the group takes all of the extra clauses while each of the other Path Processors take one each of the clauses towards the right of the tree.

This partitioning is a right biasing with the assumption that the left hand clauses amount to less work cumulatively than each of the branches towards the right. If there are exactly the same number of Path Processors as clauses then each Path Processor picks the clause associated with its unique identifier (U). The group count (G) is changed to reflect that only one Path Processor will be proceeding from this point. In the case where there are more Path Processors than choices, more than one Path Processor will need to be assigned to at least one of the choices. As this strategy is right biased, the branch chosen for all of the remaining Path Processors (after each clause has been assigned to a single Path Processor) is the rightmost. The U and G variables are set to properly reflect the new arrangement of Path Processors to branches.

In the following algorithm the setmax variable contains the value of the argument of the Setmax instruction. It defines the maximum number of possible choices or clauses within this set. This algorithm is only entered if the group count number (G) is greater than one. When only one Path Processor is executing within a partition a standard left-to-right depth-first backtracking search is performed.

```
int partition_right(setmax, p_lastclause)          /* right biased partition */
  int setmax, *p_lastclause;
{
    int firstclause, extra;

    if (setmax > G){                             /* more clauses than processors */
        extra = setmax - G;
        if (U == 1) {
            firstclause = 1;
            *p_lastclause = extra + 1;
        } else {
            firstclause = U + extra;
            *p_lastclause = firstclause;
        }
        G = U = 1;
    } else {                                                 /* setmax <= G */
        if (U < setmax) {
            firstclause = U;
            *p_lastclause = firstclause;
            G = U = 1;
        } else {
            firstclause = setmax;
            *p_lastclause = firstclause;
            G = G - setmax + 1;
            U = U - setmax + 1;
        }
    }
    return(firstclause);
}                                                   /* end partition_right */
```

A PP enters the algorithm with the intention of supplying values for the firstclause and the *p_lastclause variables. Firstclause is the number of the clause or branch which will initially be explored by the PP. The *p_lastclause variable contains the final clause which has been assigned to this Path Processor. Both of these values represent clauses which are contained within the same set. The firstclause value is used to determine an offset from the setmax instruction which will properly pick that numbered clause. Setmax is the instruction in the loaded code which heads the list of clauses within the set. The offset from the setmax instruction is immediately placed in the Prolog program counter. The *p_lastclause value is the clause in the set to which the Path Processor will eventually backtrack. This value is placed in the oracle structure which creates a limited choice point.

Each time a retry is performed the *p_lastclause value which is stored in the oracle structure is compared to the current clause number being executed. This clause number is held as an argument of the onum instructions. If this clause number is less than or equal to the *p_lastclause value stored in the oracle structure, that clause is explored. The usage of limited choice points involves separate sections of code being executed when any of the onum instructions onumtry, onumretry, onumtrust, and onumsing are reached. This code depends not only on the instruction to be performed, but also on the control strategy and splitting algorithm being used.

A diagrammatic example of the resulting partitions created with automatic partitioning is shown in Figure 7.10. The splitting algorithm used is partition right. The same search space is shown in each diagram as it would be partitioned by a varying number of initial Path Processors ranging from one to six. With only one Path Processor the entire tree is encircled showing that all of the search space was explored by this single PP. With more than one PP the search space is divided by the splitting algorithm in accordance to the creation of limited choice points. When a Prolog system has finished exploring its unique section of the tree it becomes idle and reports back to the Controller. When all of the Path Processors have become idle, the search is complete.

A more detailed analysis showing the changes in variables U and G is given for the initial state of six Path Processors searching the tree. Figure 7.11 labels the internal nodes in a breadth-first manner. Table 7.1 shows the changes which occur to the U and G numbers as automatic partitioning with the partition right splitting algorithm is executed. The numbers in the table are the values for the variables after each of the selected nodes from Figure 7.11 has been reached.

Where entries in Table 7.1 have been left empty that PP never encounters the specified node. At the beginning of execution all PPs are given a unique Global Server Number which is used to initialise the U variable. In all cases we are assuming that this number is equal to its Path Processor number. For example, the Prolog process called Path Processor 1 (PP1) has its unique identifier U initialised to the value 1. The group count (G) in the example is initially equal to six as there are six initial Path Processors starting exploration from the root of the search space. Entries in the table refer to the U and G numbers after the Path Processor has encountered the specified node. Notice that as soon as a Path Processor reaches the condition of U and G being equal to one, these variables do not change value.

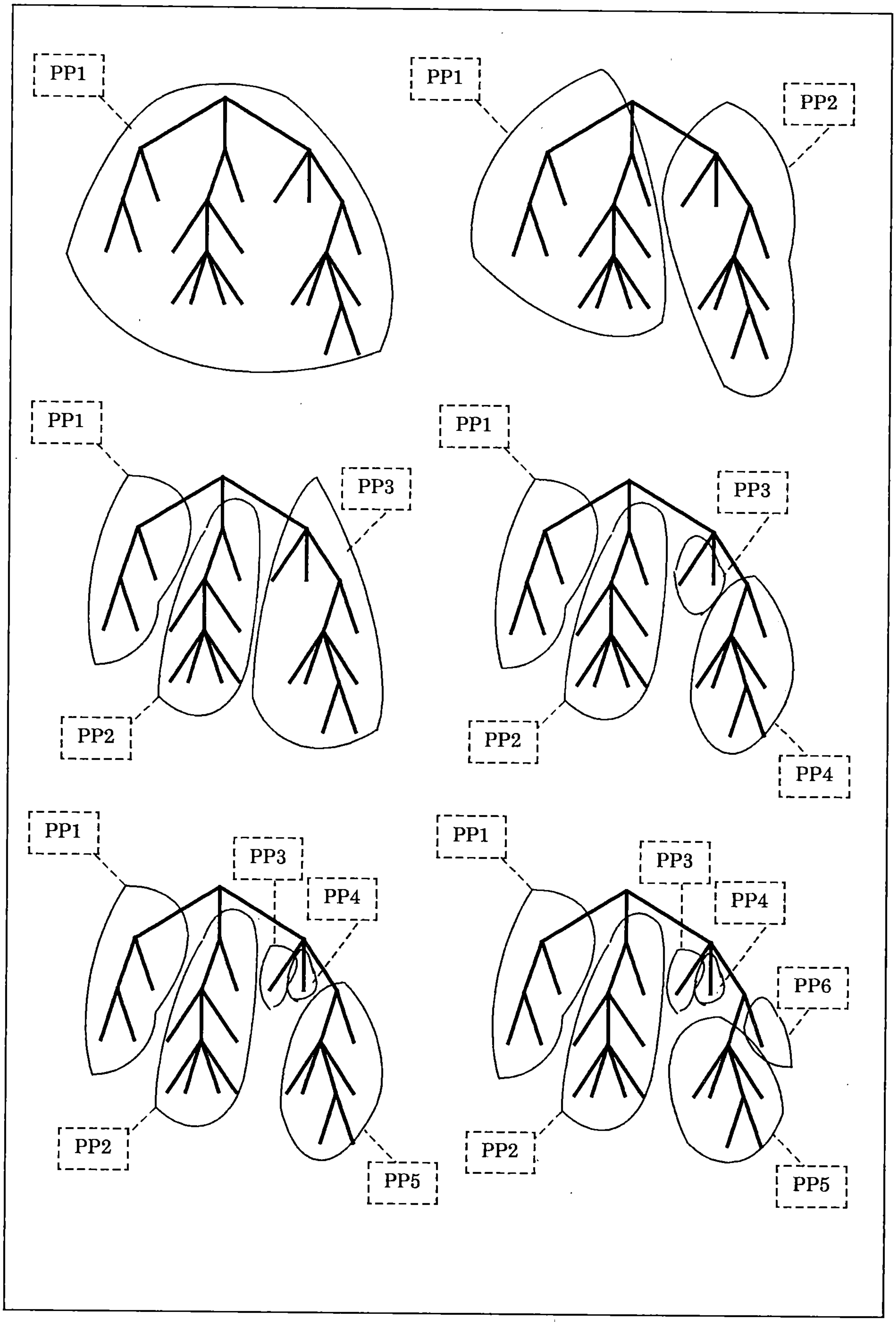


Figure 7.10 Right Biased Splitting

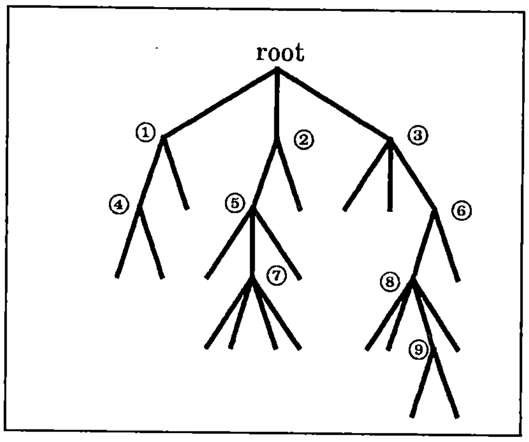


Figure 7.11 Node Labelling

Node	PP1		PP2		PP3		PP4		PP5		PP6	
	U	G	U	G	U	G	U	G	U	G	U	G
start	1	6	2	6	3	6	4	6	5	6	6	6
root	1	1	1	1	1	4	2	4	3	4	4	4
1	1	1										
2			1	1								
3					1	1	1	1	1	2	2	2
4	1	1										
5			1	1								
6									1	1	1	1
7			1	1								
8									1	1		
9									1	1		

Table 7.1 Automatic Partitioning Example

Central splitting partitions the search tree in a less biased manner than the partition right algorithm. If there are more branches at a choice point than PPs, the extra branches are distributed among the lower numbered PPs. If there are more Path Processors than clauses, the leftmost branches have extra Path Processors assigned to them. The partition central algorithm is shown below.

```
int partition_central(setmax, p_lastclause)
  int setmax, *p_lastclause;
{
    int firstclause;

    if (setmax == G){
        firstclause = U;
        G = U = 1;
        *p_lastclause = firstclause;
    } else if (setmax > G){
        int div, rem, maxnum;

        div = setmax / G;                                /* min clauses any proc gets */
        rem = setmax % G;                               /* num procs that get 1 extra */
        maxnum = div + 1;
        if (U <= rem) {                             /* these procs get maxnum clauses */
            *p_lastclause = U * maxnum;
            firstclause = *p_lastclause - (maxnum - 1);
        } else {                                       /* these procs get div clauses */
            *p_lastclause = U * div + rem;
            firstclause = *p_lastclause - (div - 1);
            if (div == 1) *p_lastclause = firstclause;
        }
        G = U = 1;
    } else {                                                        /* setmax < G */
        int div, rem;

        div = G / setmax;                                 /* minimum procs per clause */
        rem = G % setmax;                               /* clauses that have extra proc */
        firstclause = U;
        U = 1;
        while (firstclause > setmax) {
            firstclause -= setmax;
            U++;
        }
        G = div;
        if (firstclause <= rem) G++;
        *p_lastclause = firstclause;
    }                                                           /* end of if cases */
    return(firstclause);
}                                                       /* end partition_central */
```

Divisions of the search space using the partition central algorithm with from one through six initial Path Processors is shown in Figure 7.12.

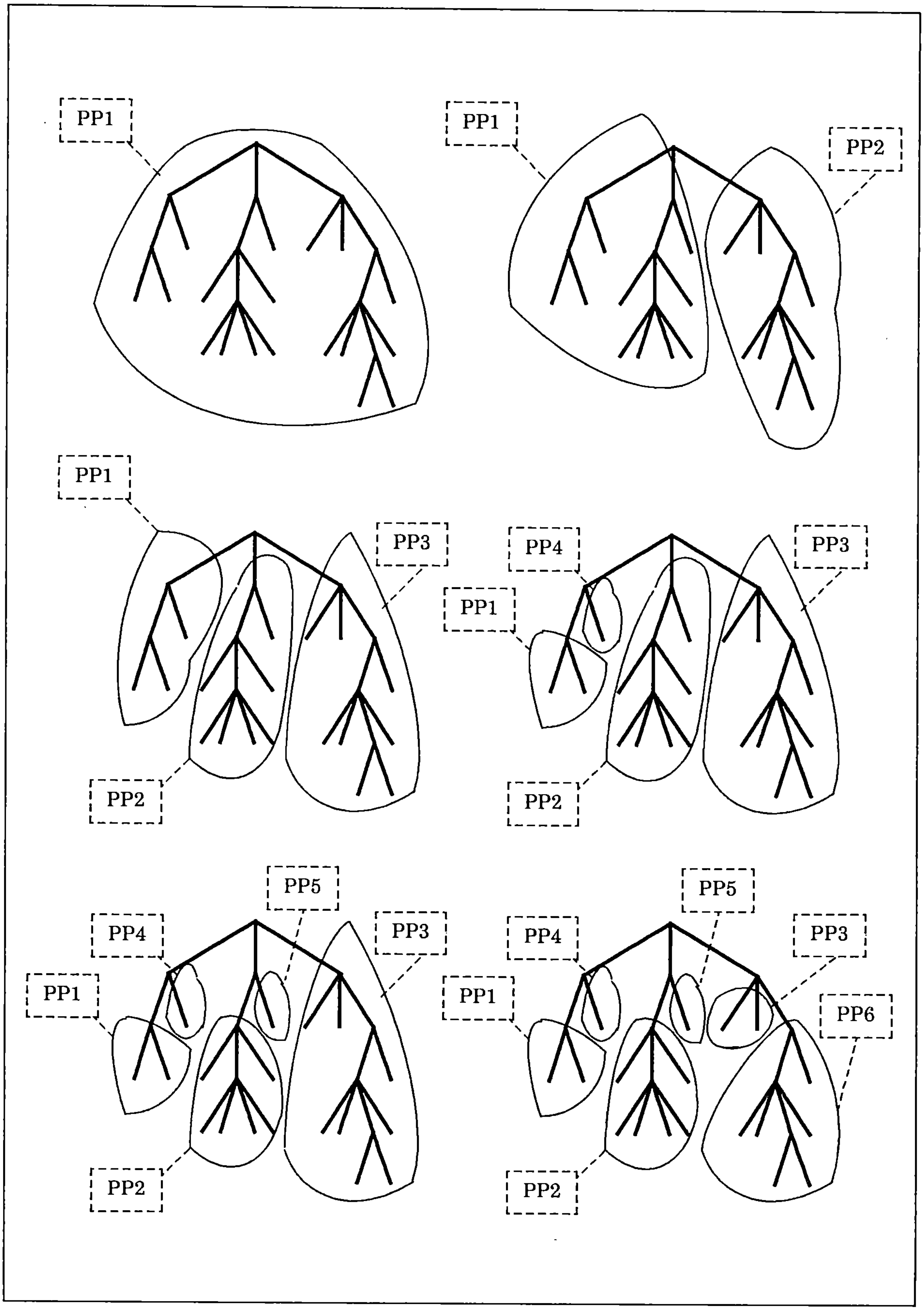


Figure 7.12 Central Splitting

7.9 Reassigning Jobs

A summary of the major features of automatic partitioning is contained in the following statements:

(1) All Prolog Systems (or equivalently Path Processors) start at the root of the search tree.

(2) Each Prolog System automatically finds a unique portion of the original tree to explore.

The second feature is obtained by manipulating the two variables G and U. Variable G being the total number of Prolog systems in a group and U the unique number given to each Prolog system. When the number of Prolog systems in a group was equal to one, a Prolog system was exploring a portion of the original search space by itself. When it had completed exploration of its section of the original tree the Prolog system became idle and did not receive new work to perform. The reassign jobs strategy is an extension of automatic partitioning which allows idle Path Processors to be assigned new work to perform. This work entails the exploration of new unique sections of the original search tree.

Reassigning jobs to idle Path Processors can be seen as a generalisation of feature number one of the automatic partitioning strategy summarised above. Instead of starting a group of Prolog systems only at the root of the search space, a group can be given any arbitrary node as a starting point. An oracle is used to describe the path from the root of the search space to this starting point. This oracle along with the group count (G) and unique identifier (U) is given to each participating Prolog system. This information is all that is needed to perform automatic partitioning from any arbitrary node.

The tree in Figure 7.13 is used as an example of starting automatic partitioning at a node other than the root of the search space. Assume we have six Prolog systems which are requesting additional work to perform. The tree has been searched down to the node pointed at by the arrow (see Figure 7.13) so we wish to begin an automatic partitioning strategy starting from this location. Each of the six Prolog systems is given the oracle [1011] to follow along with two additional pieces of information; the group count (G) and a unique identifier (U). In this case G is equal to six and so the unique identifiers range from one to six. If the partitioning strategy being used is right biased, we now have a situation equivalent to that demonstrated in Figure 7.10 and Table 7.1. The only difference is the location within the search space where automatic partitioning is initiated.

Two questions that have not been answered in the previous discussion are:

(1) Where did the available Prolog systems come from?

(2) What nodes are potential choices from which automatic partitioning can be initiated?

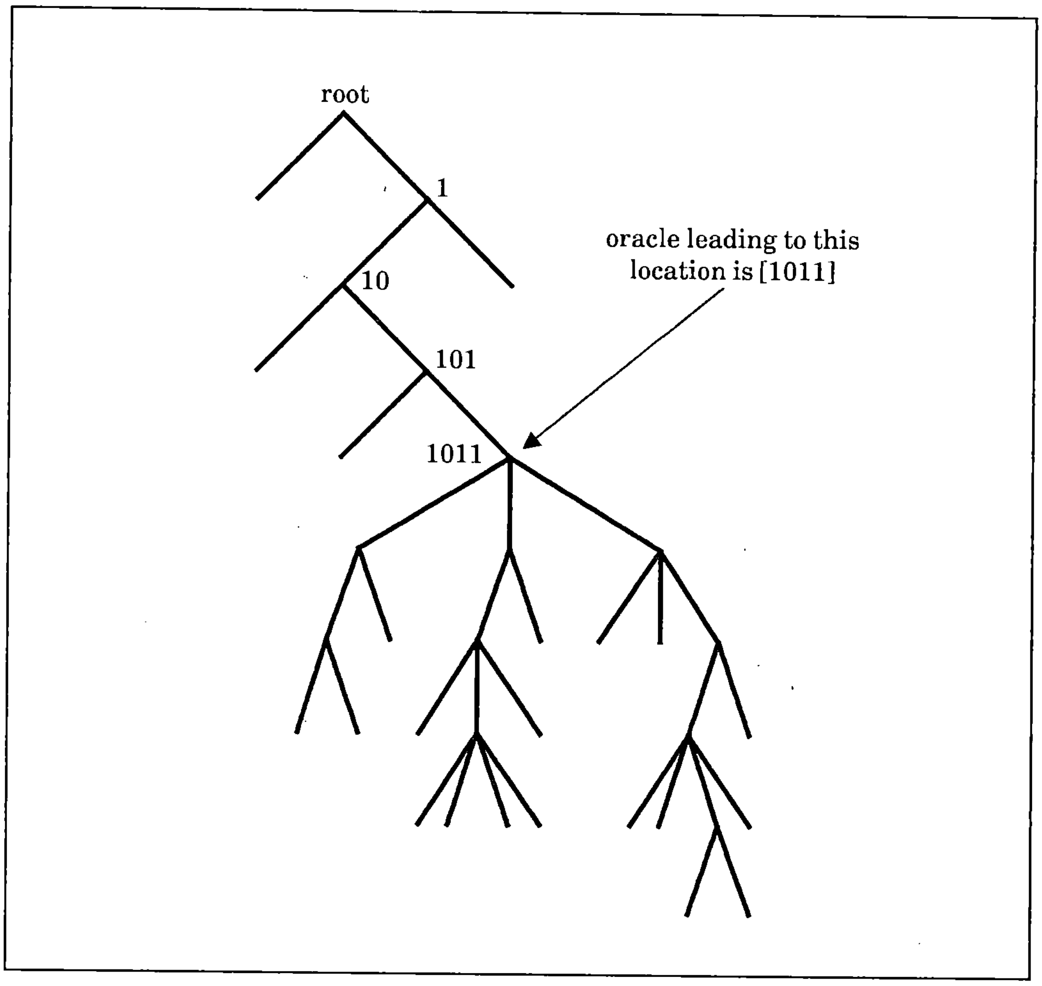


Figure 7.13 Automatic Partitioning Starting From an Arbitrary Node

The first question can be answered simply by mentioning that all available Prolog systems are maintained in the idle process queue. If the idle process queue is empty then there are no Prolog systems available to perform new work. In the majority of cases the reason a Prolog system appears on the idle process queue is that it has completed the exploration of its unique portion of the search tree. The only other time that a Prolog process joins the idle process queue is when that process has just been initialised. For most Prolog processes this occurs at the beginning of a Delphi run. Each Prolog started at the outset sends the need new job message and receives the initial G and U parameters; automatic partitioning is then begun starting at the root of the search space.

There is a mode for running Delphi which permits Prolog processes to be started and used at any time throughout the processing. The initialisation parameters sent to each Prolog system at the start of the Delphi run contain the original G value. If any additional Prolog processes are started after this number is distributed, the process must wait on the idle process queue until new work can be given it. It will not participate in the initial automatic partitioning from the root of the search space. The Prolog process (and all processes on the queue) will participate in the next automatic partitioning group requested after it has been received on the queue. This group will begin automatic partitioning from some node other than the root node.

The second question asks how and when these idle Prolog processes receive new sections of the search tree to explore. Before this question can be answered, the concept of a *check-in* must be described.

When a process is uniquely searching some portion of the original tree, it has G and U values equal to one. Under these conditions when a branching point is reached, the Prolog can *check-in* to the Controller process to request a distribution of the workload. The check-in consists of sending an oracle to the Controller to describe that Prolog's position within the search space. The Controller then sends back an integer telling the Prolog the number of idle Path Processors in the idle process queue. If this number is zero then the check-in has been unsuccessful. The Prolog has no other processes to split the work with and must therefore continue searching the tree on its own. If the number sent back by the Controller is greater than zero, then the Prolog which has checked in becomes one of a group of processes which will automatically divide the work. The group count G will be equal to the number of processes in the queue plus one (the Prolog which did the check-in). This group can then split the work of exploring the tree starting from the branching point sent as an oracle to the Controller.

The final question to answer is when the check-ins to the Controller occur. It has been mentioned that a Prolog will only check-in when G and U are equal to one. If this were the only constraint on performing a check-in then check-ins would occur at every branching point that the Prolog reaches by itself. To provide additional flexibility into the system a *check-in interval* is defined.

A *check-in interval* limits the frequency at which check-ins will occur. It sets the minimal change in depth that must occur before a check-in will be performed. For example, suppose that the check-in interval is set to one. Check-ins will be performed at every change in depth of one level. This means that check-ins will occur at every branching point where a Prolog system is searching on its own. The number of check-ins that will be performed will be large even for small search spaces. If the check-in interval is set low then the check-in frequency will be high. The inverse is also true. If the check-in interval is set high then the number of check-ins actually performed (the check-in frequency) will be low.

Figure 7.14 is a demonstration of the reassign jobs strategy with three participating Prolog processes and the check-in interval set to one. The partitioning strategy used will be the partition right algorithm. Since the events in a Delphi run are asynchronous, Figure 7.14 shows just one of a number of possibilities for this reassign jobs example with three initial processes and a check-in interval initialised to one.

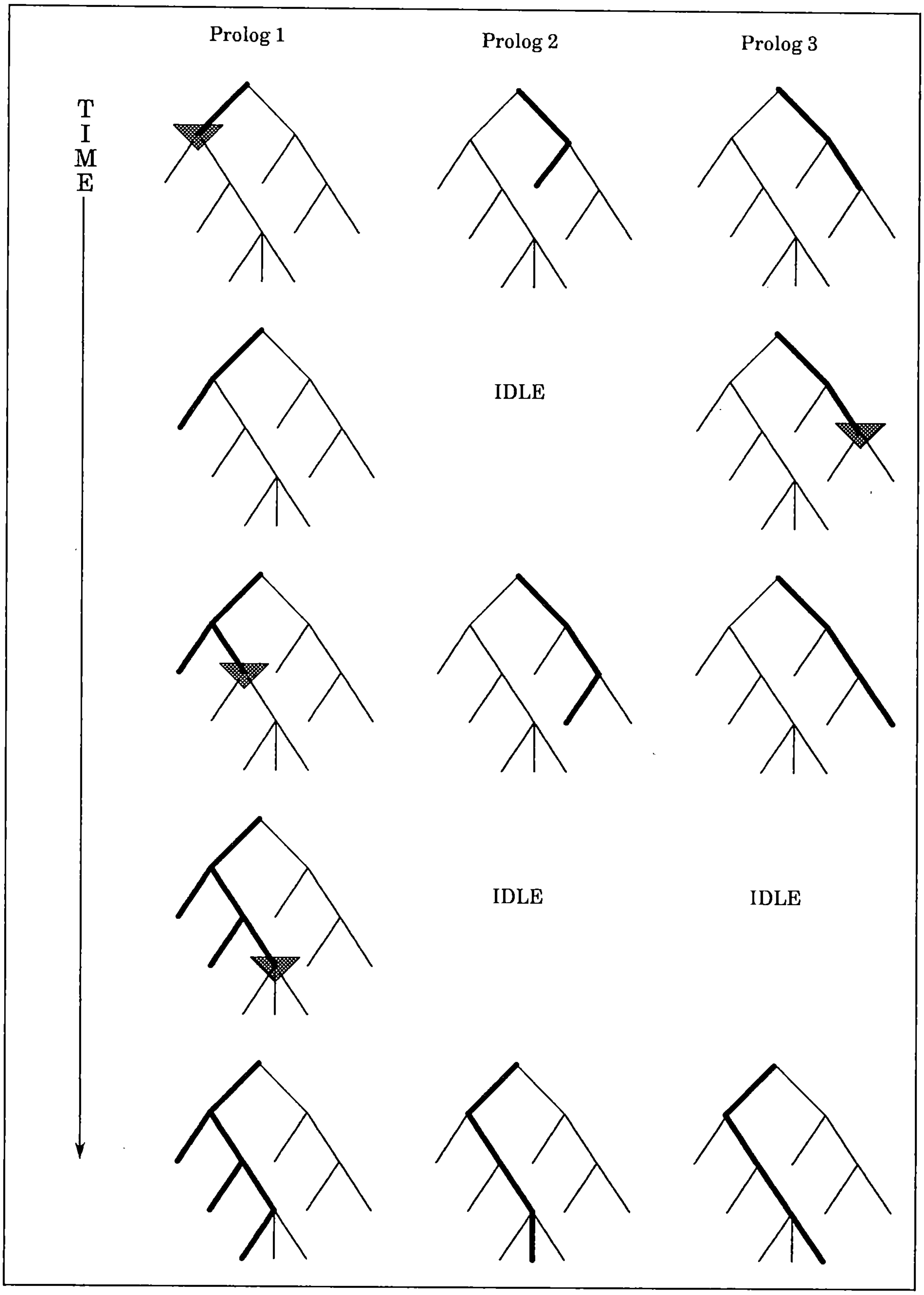


Figure 7.14 Reassign Jobs Example

For simplicity of this discussion, we will consider Figure 7.14 as five time frames. A frame consists of three Prolog systems at the same instant in time. In frame number one, all of the Prologs start at the root of the search space and automatic partitioning takes place. Prolog 1 (PP1) takes the left-hand branch while the other two (PP2 and PP3) take the right-hand branch. PP1 realises that it is on its own and with a check-in interval equal to one, it checks in to the Controller. A check-in is shown by the shaded triangle. There are no idle Prolog processes for PP1 to split the choice point with and so the check-in is unsuccessful. PP1 will continue to search the tree on its own. PP2 and PP3 both reach the node located by oracle [1] and so they automatically split the choice point with PP2 going to the left and PP3 to the right. No check-ins are performed at node [1] since the G number was equal to two.

In frame number two PP1 has continued to search the tree on its own after the unsuccessful check-in. It reaches a leaf node [11] and there is still more tree to explore (for example [01] since the branching point at [0] was not divided) so it will backtrack. PP2 has become idle since there is no more work (portion of the tree to explore) for it to do. PP3 reaches a branching point on its own and performs a check-in. There is one idle process (PP2) and so these two will split the choice point at [00].

In frame three, PP1 has backtracked, arrived at [01], and performed a check-in to the Controller. There are no idle processes and so it continues the solo search. Because of the successful check-in performed by PP3, the choice point located at [00] is split between PP2 and PP3. PP2 takes the left-hand branch and PP3 takes the right.

In frame number four PP1 has explored [010], backtracked to [011] and performed a check-in to the Controller. This check-in is successful. PP2 and PP3 have both become idle after exhausting their portions of the tree. Because of the check-in by PP1, all three of the Prologs will participate in splitting the branching point described by oracle [011]. Since the branching factor is equal to the number of Prolog processes, each process explores one of the branches. Frame five shows each Prolog system exploring its own unique branch. If there were a sixth frame it would show all three Prolog processes idle and the search would then be complete.

Chapter 8 Results

Benchmarking of the Delphi machine began with the new implementation using 4.2BSD sockets for interprocess communication. Non-backtracking strategies had been left behind with the implementation using Amoeba-transactions-under-UNIX. The few timings made with the Amoeba implementation made it appear that non-backtracking strategies were too inefficient. Even taking into account all of the problems with the Amoeba system, the execution times were excessive. The 4-queens problem was taking over half an hour to run on a single processor. This problem was known to run in under two seconds on a sequential Prolog. Alshawi and Moran [1988] came to the same conclusion that non-backtracking strategies are too inefficient. The results they show for their Delphi implementation are only for backtracking control strategies; the same is true for the results presented in this chapter. All of these results were obtained from two general backtracking control strategies. The first backtracking strategy implemented used *automatic partitioning*. This strategy was to become the initial mode for all future backtracking strategies. Next the more general *reassign jobs* strategy was developed and implemented. The results in this chapter show applications of these two strategies to various Prolog programs. Sources for the benchmark programs can be found in Appendix 8b.

8.1 Initial Results

The problem chosen as a first benchmark was the 8-queens problem. This problem is used as a benchmark for parallel Prolog [Shapiro 1989] and other parallel language and machine implementations [Finkel and Manber 1987]. The problem does not take long to execute (under ten minutes) and the number of answers is large (ninety-two). This problem would be a good test to ensure that the system was reporting all of the results properly to the global log file. A check-in number was arbitrarily chosen to be fifty and a shell script used to run the benchmark from one to the maximum number of processors available which was twenty. Figure 8.1 shows the first results from the 8-queens problem being run on from one to twenty μVAX machines. The reassign jobs strategy and *partition right* algorithm were used with a check-in interval of fifty. At every change of depth greater than or equal to fifty levels each Prolog system checks with the Controller to see if there are any idle processors waiting for work. The execution times are in seconds as are all of the timings given throughout this document.

Figure 8.1 shows an anomaly (encircled) when eight processors are configured for execution of the 8-queens problem. These results were obtained on a weekday afternoon. The problem was run again early in the morning to test whether this irregularity is a function of the 8-queens problem or some symptom of the network. This avoided contention with other users for host machine cycles and access on the Ethernet. The results from this early morning run are shown in Figure 8.2.

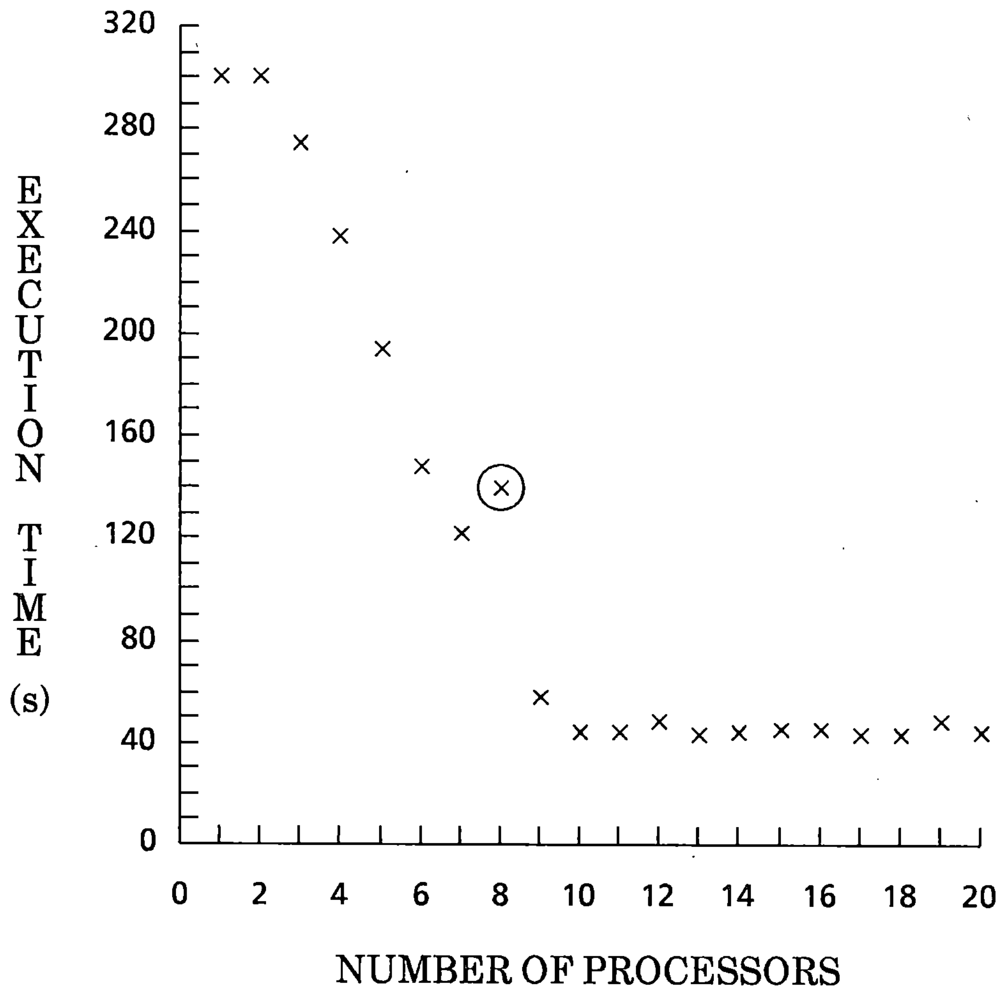


Figure 8.1 8-Queens Problem - Afternoon Run

An interesting feature of both Figure 8.1 and Figure 8.2 is that they are only able to use ten out of the twenty available processors. Four hypotheses for this are:

(1) The communication overheads with over ten processors start to reduce any of the speed ups obtained by increasing the number of processors.

(2) Forty to fifty seconds represents the minimal load and initialisation time for this problem. The initialisation overheads cannot be reduced below this level.

(3) All of the available OR-parallelism in this problem has been exploited. There is a limit on the amount of OR-parallelism available in a program. For the 8-queens problem this limit is reached when ten or more processors are configured for a run.

(4) This is a feature specific to the N-queens problem. Something about the search space of the N-queens problem causes this characteristic levelling out.

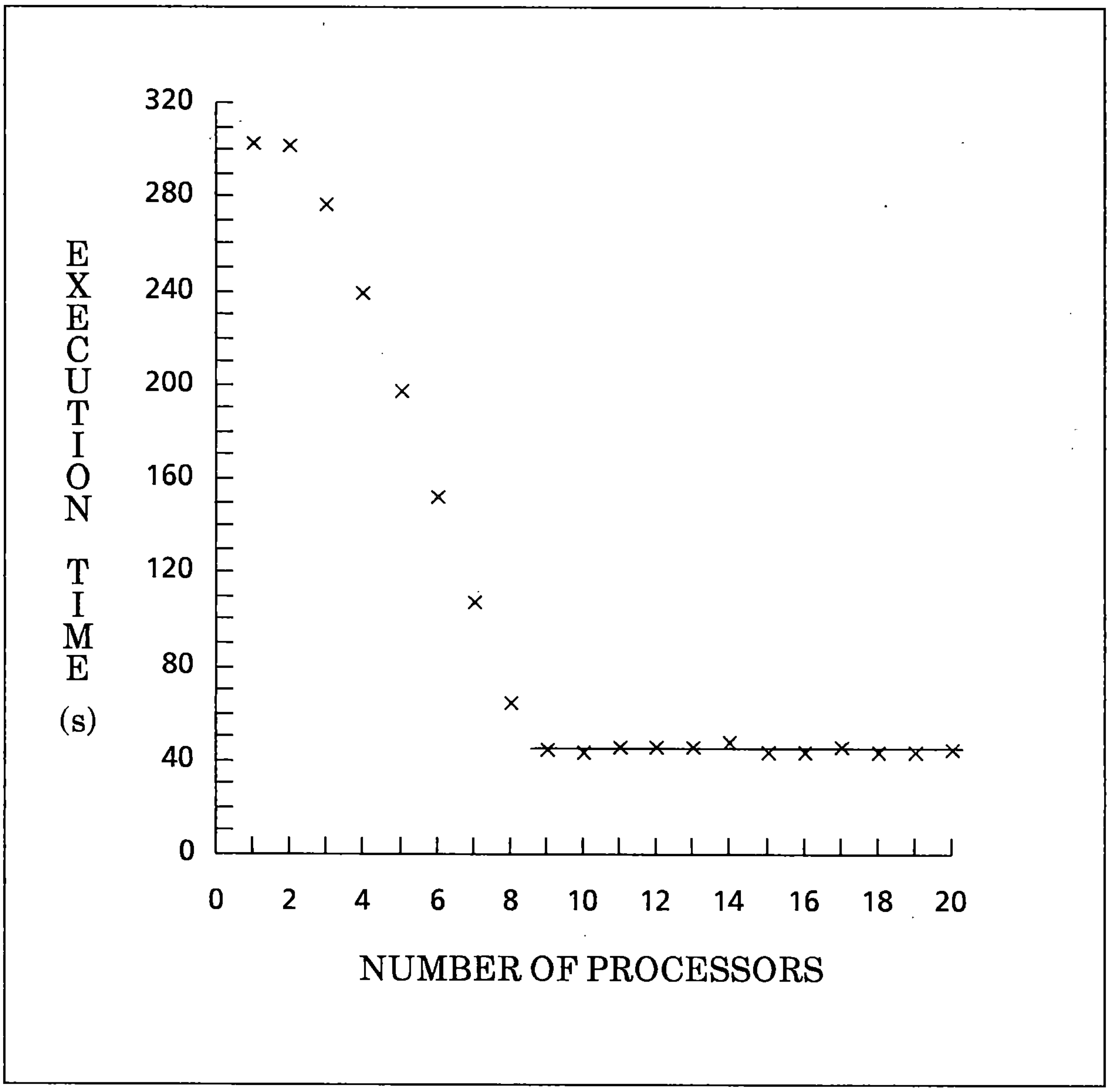


Figure 8.2 8-Queens Problem - Early Morning Run

If communications costs were the root of the problem, we would not expect the curve to level out. This would only happen if the communication costs were balancing the speed ups so that the end result was constant. If communication costs were the dominant overheads in this problem we would expect the values to increase after the minimum value of around forty seconds is reached. Since this does not happen it suggests that the communication costs are not interfering significantly with the execution of the problem. Figure 8.2 demonstrates that even when double the number of processors needed to reach the minimum time are configured, the execution time does not increase. If communication overheads were the major factor in the inability to use more than ten processors we would expect a plot looking something more like that in Figure 8.3.

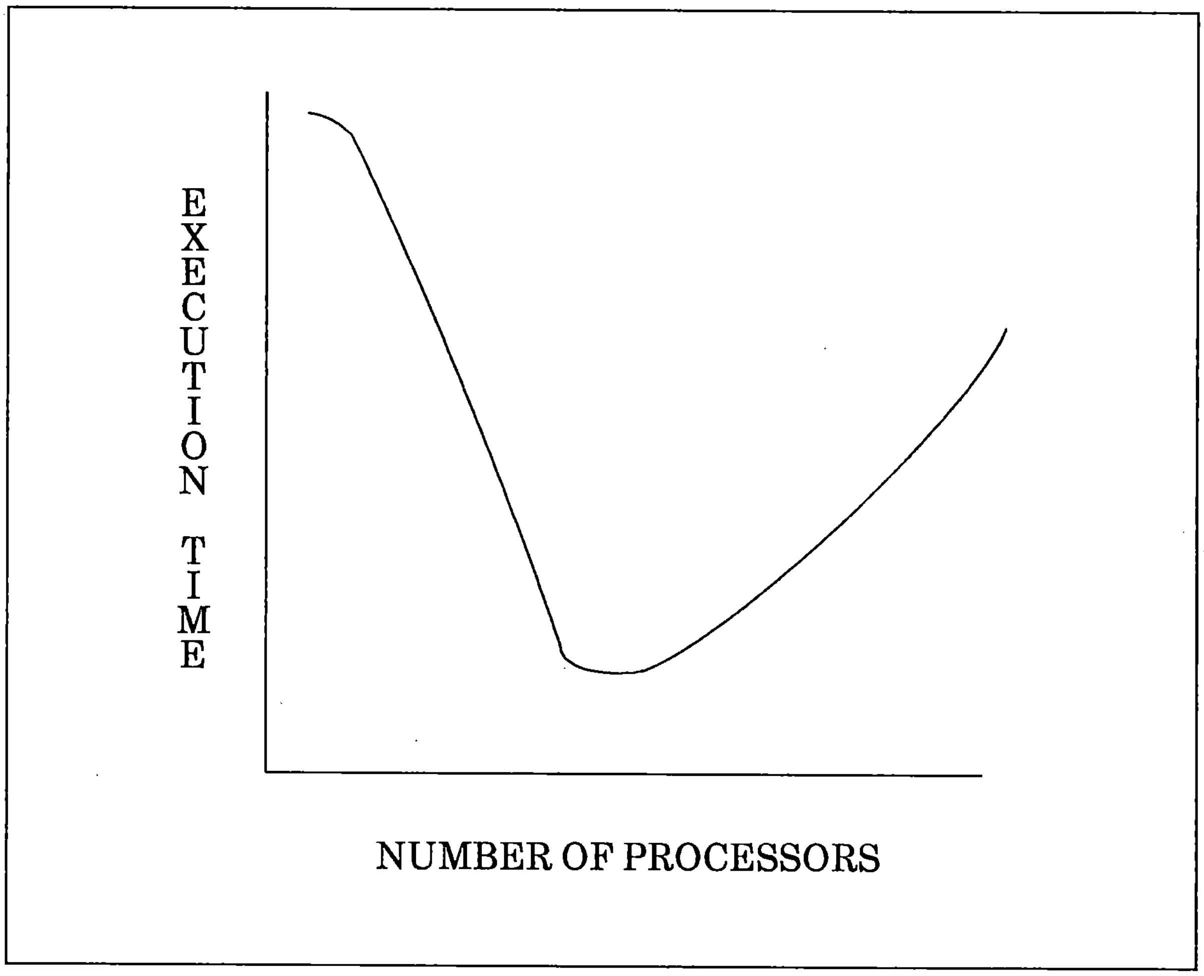


Figure 8.3 Hypothetical Plot Demonstrating Communications Overheads

The second hypothetical explanation for why the 8-queens plot (Figure 8.2) levels out is that some minimal initialisation time has been reached. This initialisation time includes the starting of Prolog processes over the network, loading of the compiled program and reading in the system start up files. If it is true that the minimal initialisation time for this problem was reached, then there will be some characteristic loading and initialisation time for every problem tested. If the number of processors is sufficient to bring the execution time down to a minimal time required for initialisations, the plots will level out at this point. One way to test this is try more problems and see if the characteristic levelling out occurs.

In running numerous problems it was seen that the initialisation times were very similar. The length of the compiled code varies very little so the time it takes to load the code is comparable in each case. The time it takes to initialise the Prologs on the various host machines is not dependent on the problem to be executed, so these times will also be similar. Initialisation time is a function of the load on each of the host machines and on the network. It turns out that this time is not affected appreciably by changes in the Prolog program. The initialisation times vary from nearly zero seconds to a little under a minute, depending on the load on each of the host machines, but not on the problem to be executed. Because the initialisation time is dependent on the network load, problems were benchmarked when the network load was very light (in the middle of the night). We would expect these results to have similar initialisation times. If the level portion in the 8-queens plot (Figure 8.2) is indicative of an absolute minimal initialisation time then other problems would show this same bottom limit. Other problems did not show a common bottom limit. This means that the supposition regarding initialisation time is insufficient to explain our measurements.

Table 8.1
The 10-Queens Problem Executed on From One to Twenty µVAXes;
Data for the Plot in Figure 8.4

Number of Processors	Execution Time in Seconds
1	8137
2	6723
3	6335
4	5183
5	4477
6	3811
7	2805
8	1766
9	1542
10	1394
11	1010
12	940
13	855
14	784
15	680
16	534
17	523
18	501
19	468
20	450

The third hypothesis is that the levelling out of the 8-queens plot (Figure 8.2) is caused by the upper limit of OR-parallelism (as exploited by the splitting strategy used) being reached. There are several ways to test this hypothesis. The first is to run a larger version of the N-queens problem to see if any more processors are used. We would expect that the 10-queens problem has more OR-parallelism than the 8-queens problem. Assuming that this is correct and that the limit of the OR-parallelism available in the 8-queens problem had been reached, we should see an increase in the number of processors used when running the 10-queens problem. For the 8-queens problem the number of useful processors was about ten. For the 10-queens problem we would predict that more then ten processors would be used. The 10-queens problem was attempted next with the results shown in Table 8.1 and Figure 8.4.

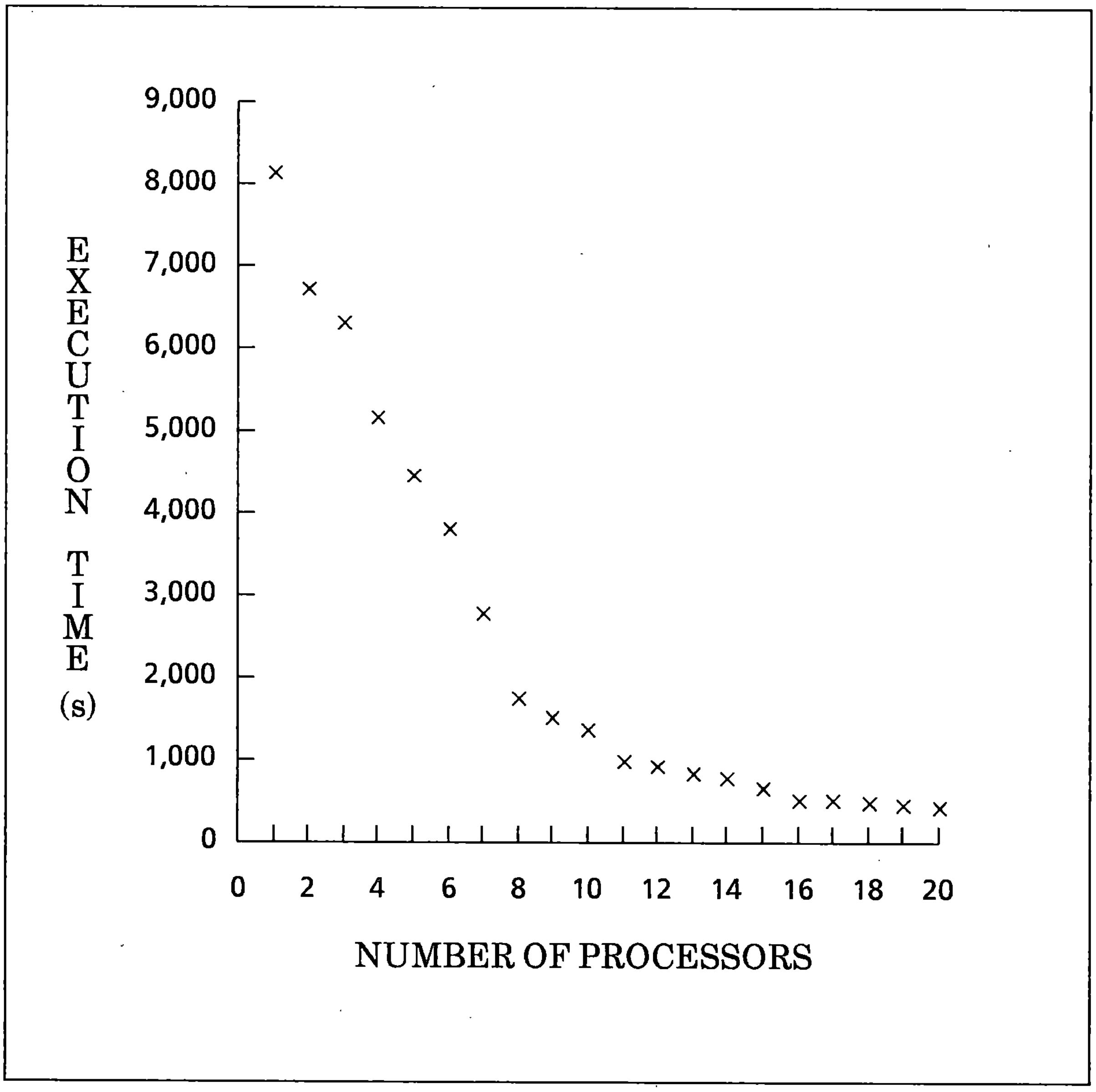
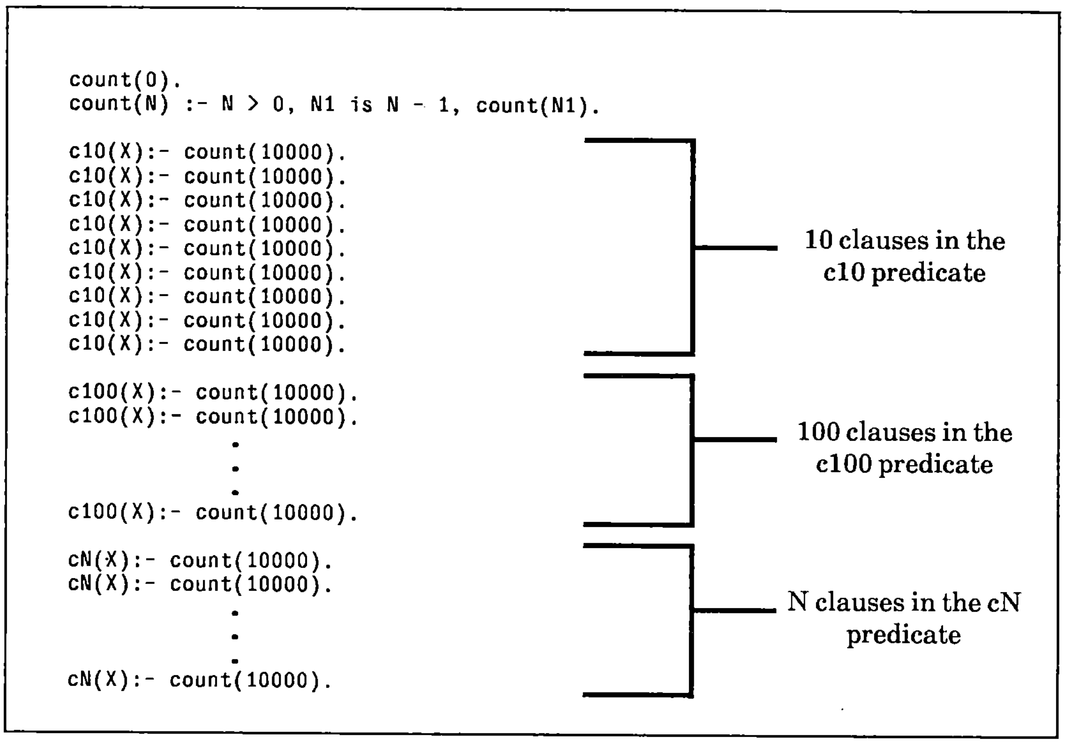


Figure 8.4 10-Queens Problem on μVAXes

It is difficult to see from the plot whether there is a levelling off at a particular number of processors. What is easy to see is that the rate of change in the execution time is decreasing as the number of processors increases. It looks as if the times are approaching some minimal value. Table 8.1 shows the execution times used to produce the plot in Figure 8.4.

From Table 8.1 it appears that the 10-queens problem uses all of the available host machines. To obtain a lower limit we would have to run the problem on a larger number of processors. As there were only twenty available this could not be done so a smaller problem (9-queens) was run. The 9-queens problem did show a minimal value being reached when the configuration contained at least fifteen processors. This suggests that the third explanation is more accurate; the limit of OR-parallelism is being reached in the 8-queens and the 9-queens problems when fewer than twenty processors are configured.

There is another way to test whether the amount of OR-parallelism in a program is proportional to the number of processors that can be effectively used by Delphi. This is to increase the amount of OR-parallelism by increasing the number of clauses in a predicate. Figure 8.5 is an example of a contrived problem written to test the usage of processors. The program is named `ortest`.



```
count(0).
count(N) :- N > 0, N1 is N - 1, count(N1).

c10(X):- count(10000).
c10(X):- count(10000).
c10(X):- count(10000).
c10(X):- count(10000).
c10(X):- count(10000).
c10(X):- count(10000).
c10(X):- count(10000).
c10(X):- count(10000).
c10(X):- count(10000).

c100(X):- count(10000).
c100(X):- count(10000).
              .
              .
              .
c100(X):- count(10000).

cN(X):- count(10000).
cN(X):- count(10000).
            .
            .
            .
cN(X):- count(10000).
```

10 clauses in the c10 predicate
100 clauses in the c100 predicate
N clauses in the cN predicate

Figure 8.5 Ortest Program

Ortest was run on Delphi with the queries c10(X), c50(X) and c100(X). Plots of these runs are shown in Figure 8.6. Results of the best Delphi run are compared to C-Prolog and Cosmic Prolog (the unmodified Prolog system used to develop the Delphi Prolog system) in Table 8.2.

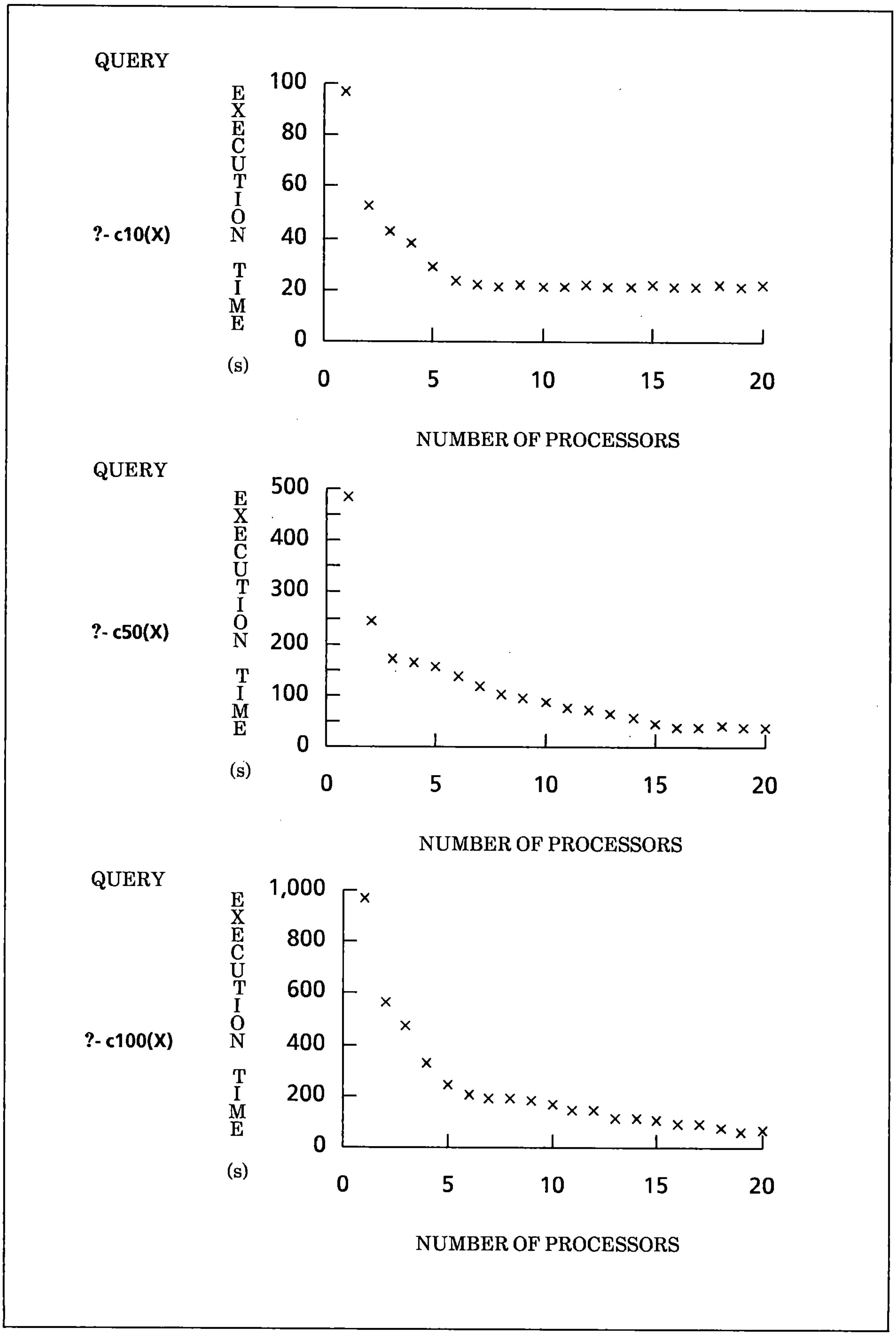


Figure 8.6 Results of Three Ortest Queries

All three of the Prolog systems shown in Table 8.2 (and throughout this chapter) were run on μVAXes. The times shown have been rounded to the nearest second. The version of C-Prolog used is Version 1.4 [Pereira 1984]. Cosmic Prolog is essentially SB-Prolog Version 2.2 with modifications to the floating point number instructions and various bug fixes. SB-Prolog or Stony Brook Prolog is a public domain piece of software developed at SUNY (State University of New York) at Stony Brook. The execution times for Delphi are for a twenty processor configuration.

Problem	C-Prolog	Cosmic Prolog	Delphi (20 processors)
c10	253	66	21
c50	1247	274	38
c100	2465	587	62

Table 8.2 Ortest - Comparison of Execution Times

8.2 No Control Communications Needed

The 8-queens problem was run with the initialised check-in interval set to different values. It was noticed that the best execution times were obtained with a very wide range of check-in intervals. Even more unusual was that when the check-in interval was set higher than the maximum depth of the tree the optimal results were still obtained. When the check-in interval is set higher than the maximum depth of the search space then no check-ins ever occur. This means that the 8-queens problem was achieving optimal results without any control communications at all.

Setting the check-in interval to be higher than the depth of the search space is equivalent to running an automatic partitioning strategy only. As soon as a processor becomes idle it is never reassigned any more work to do. It was thought that automatic partitioning would leave one or a few processors with the bulk of the work to do while all of the rest of the processors sat idle during most of the execution time. To check exactly what was happening at various check-in intervals, an extra facility was added. This was to allow logging of messages about the active or idle state of each processor. An active state occurs when a processor is busy executing the program. These active and idle times are then plotted to get an idea of how balanced the loading is on each of the host machines. Two separate plots are produced. Figure 8.7 is an example of the first type of plot.

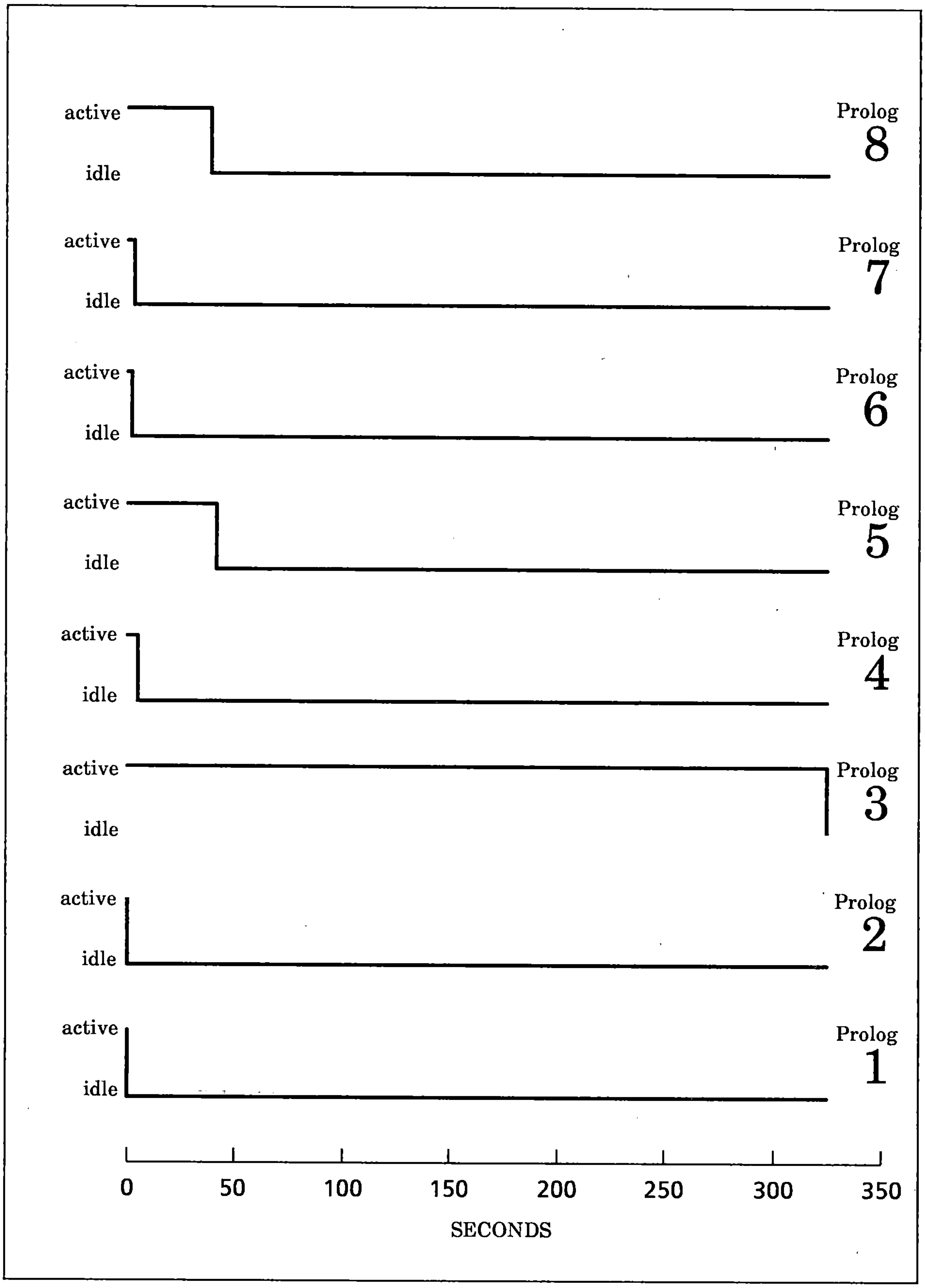


Figure 8.7 Loading for Auto Partitioning (Check-in Interval Initialised to ∞)

Eight Prologs were run on eight separate host machines to produce the *waveform* plot in Figure 8.7. The problem executed is the 8-queens problem. The right side of Figure 8.7 shows the unique identifier given to each Prolog system as it logs in to the Controller. On the left side are shown the different levels for active and idle states of each Prolog. The time on the bottom scale is in seconds. The increased time taken to perform this problem is the result of extra logging messages sent to the Controller. To produce this plot the check-in interval was set to a very high number which is much greater than the maximum depth of the search space. We say that the check-in interval is "set to infinity". When the check-in interval is set to infinity the control strategy that is being used is automatic partitioning.

Figure 8.7 shows the results from running the automatic partitioning strategy. These are the results that were expected from this control strategy. Two processors, Prolog number 1 and Prolog number 2, become idle immediately while three others only contribute a few seconds of execution time (Prolog numbers 4, 6 and 7). Three processors contribute a substantial amount of active time with one of them, Prolog number 3, performing the majority of the work. The second type of plot is a histogram showing the amount of idle and active time for each processor. Figure 8.8 corresponds with the plot in Figure 8.7. The host machine names on the left side of this plot are in the same order as the Prolog systems of Figure 8.7. Host machine *path01* executed Prolog number 1, host machine *hythe* executed Prolog number 8, and so forth.

For a comparison, the reassign jobs strategy was run with the check-in interval initialised to ten. The waveform plot is shown in Figure 8.9. This plot shows the first ten seconds of execution time. Notice that all but three of the processors have become both idle and active at least once within these ten seconds. This is to be expected when the check-in interval is set as low as ten. The active processors are checking in to the Controller so frequently that as soon as any processor becomes idle it is immediately given new work to do. The load balancing is therefore much better than with an automatic partitioning strategy. All of the processors are kept working most of the time. The problem is that there is more communication traffic making the overall execution time longer. A histogram plot of the active and idle times for the reassign jobs strategy can be seen in Figure 8.10.

With the control strategies operating as expected, the next experiment was to obtain the results over the entire range of check-ins. The range of check-ins extends from one (each processor checking in at every choice point) to seventy-eight, which is the maximum depth of the 8-queens search space. Each check-in interval was tested with between one and twenty processors. With the check-in interval set to seventy-eight there are no check-ins performed at all. Here the reassign jobs strategy can be said to be equivalent to the automatic partitioning strategy. The results are plotted for the first five check-in intervals in Figure 8.11. All of the data is presented in tabular form in Appendix 8a.

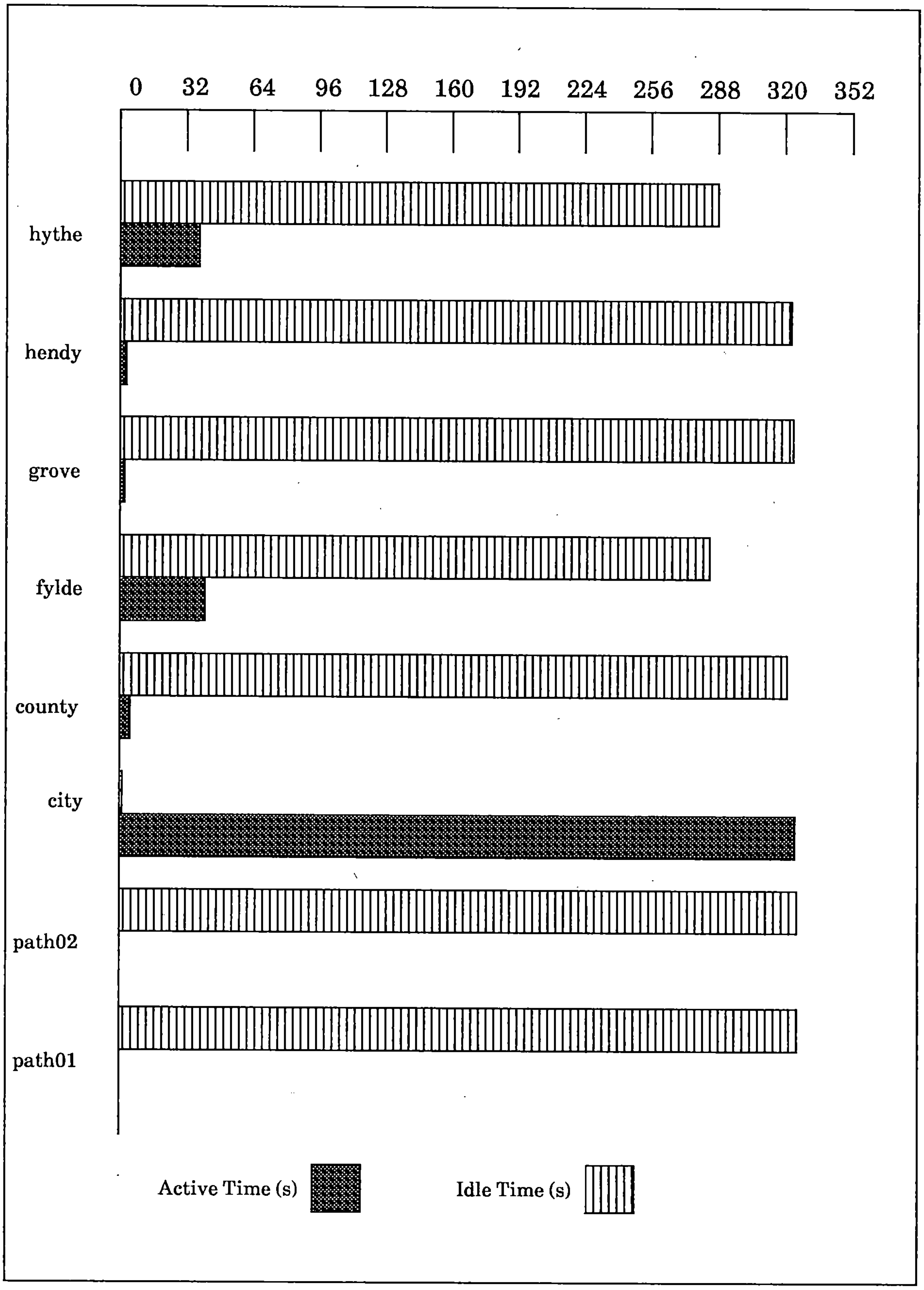


Figure 8.8 Active and Idle Time Per Processor (Check-in Interval Initialised to ∞)

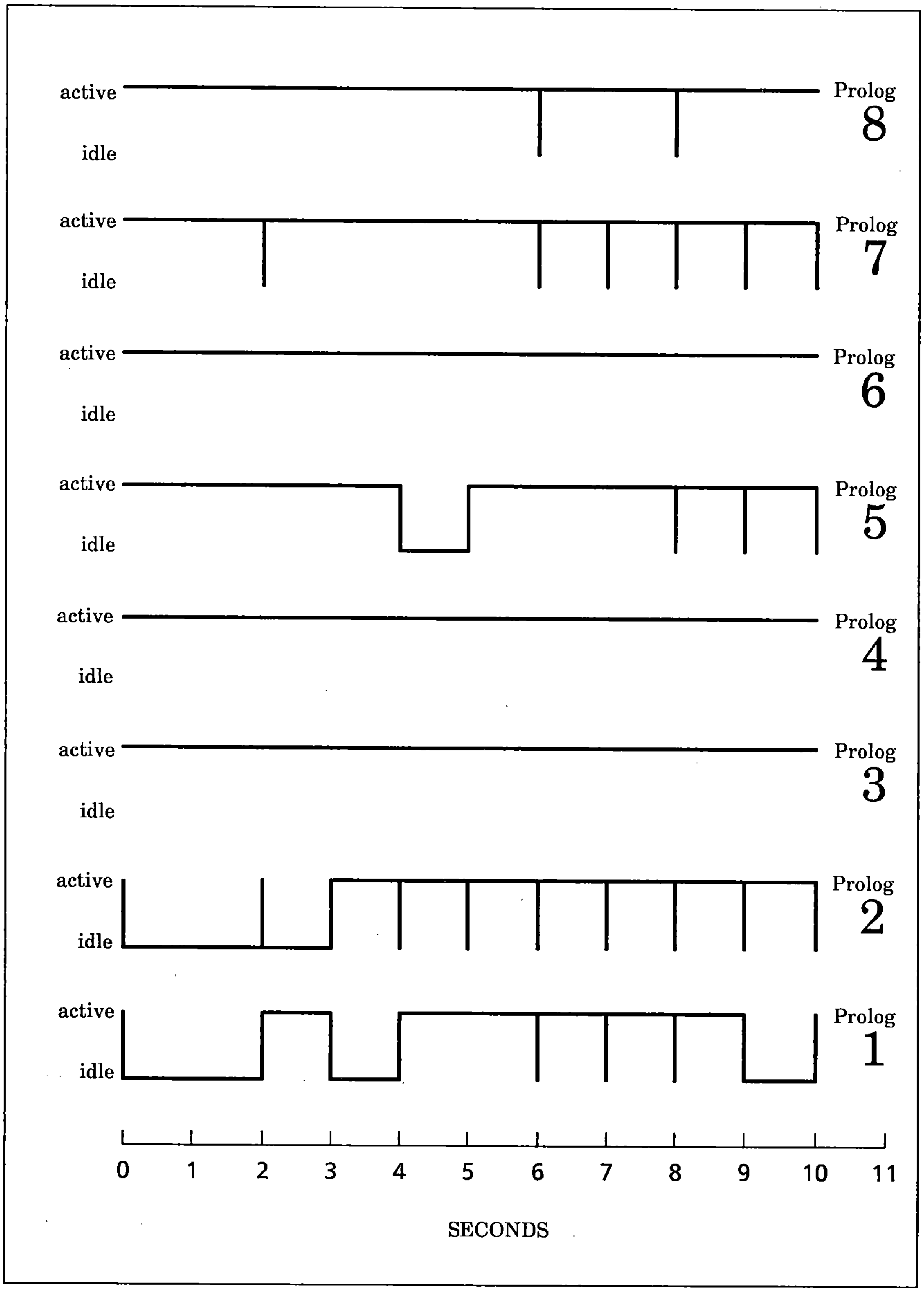


Figure 8.9 Reassign Jobs Strategy with Check-in Interval Initialised to 10

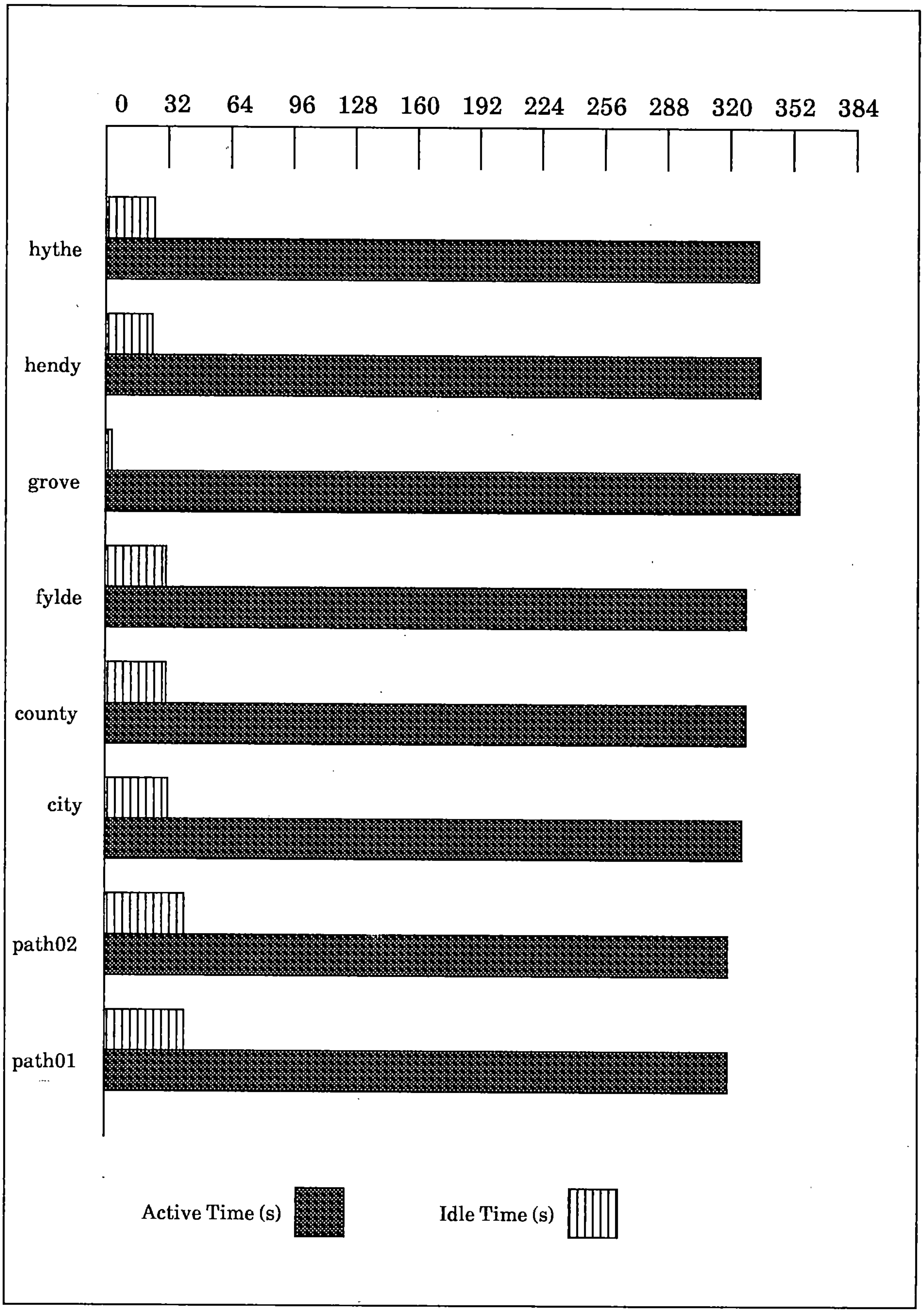


Figure 8.10 Active and Idle Time Per Processor (Check-in Interval Initialised to 10)

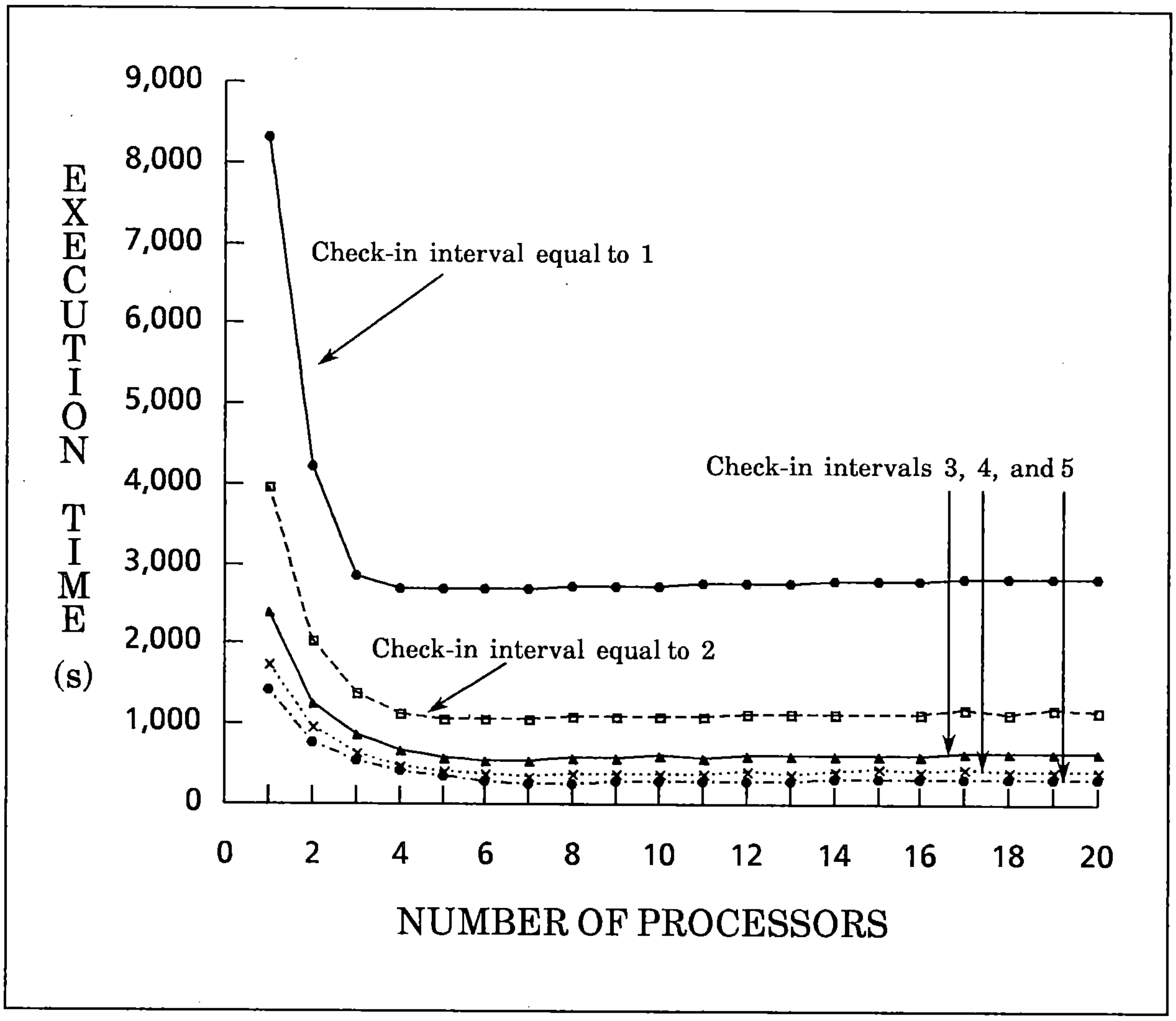


Figure 8.11 Check-in Intervals from One to Five

Many trends can be seen from the data presented in Appendix 8a. In general as the initialised check-in interval goes up for the same number of processors, the execution time goes down. If we consider all of the tables placed side by side, then a row of this large table from left to right shows a decrease in the execution time. In general as the number of processors is increased for a particular check-in interval, so the execution time decreases. Looking down a column the execution time decreases for most columns. The major exception to the inverse relationship between the number of processors and execution time is when the check-in interval is very low. Even here, the added communications do not cause a relatively great increase in the execution time. Looking down the columns of the low check-in intervals the execution time appears to level off rather than get much higher. The reason is that the number of check-ins performed is fairly constant within a column.

Initialised Check-in Interval	Number of Check-Ins Performed	Initialised Check-in Interval	Number of Check-Ins Performed
1	41182	21	988
2	19475	23	911
3	11697	24	792
4	8463	25	696
5	6711	26	620
6	5744	27	562
7	5007	28	491
8	4376	29	428
9	3708	30	330
10	3142	31	292
11	2759	32	203
12	2433	33	141
13	2230	34	102
14	2045	35	73
15	1846	36	38
16	1652	37	12
17	1483	38	16
18	1330	39 - 77	1
19	1212	78 - ∞	0
20	1058		

Table 8.3 Number of Check-ins Actually Performed for each Initial Check-in Interval

Some interesting results shown in Appendix 8a are where the check-in interval is set high. The optimal result of between forty-three and forty-eight seconds is achieved in many of these configurations. Except for a few anomalies, if the check-in interval is greater than about thirty-eight the best result is achieved when at least eleven processors are configured. These optimal results are still obtained even if the check-in interval is higher than the maximum depth of the tree. What this means is that approximately the same number of communications are being performed in all of these configurations.

A table of the number of control communications (check-ins to the Controller) that are performed when running the 8-queens problem is shown in Table 8.3. These results were obtained by counting the number of check-ins from a single processor Delphi configuration. There is a range on the number of check-ins that occur as the number of participating processors changes from one to twenty. Even runs of the same multiple processor configuration will show a variation in the number of check-ins. This is what would be expected since the processing proceeds in an asynchronous manner; there is a random element in each individual run. A simple example of where this randomness can occur is demonstrated with Figure 8.12.

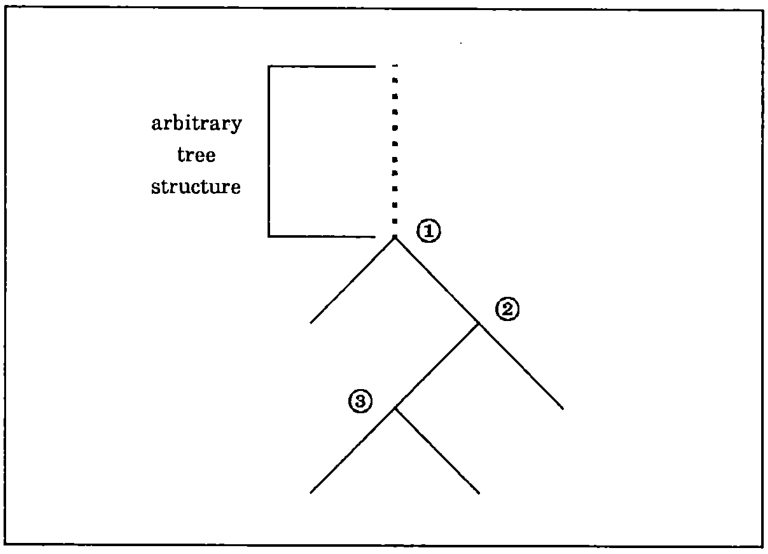


Figure 8.12 Example of a Random Component within a Run

Assume that the check-in interval has been initialised to one. A single processor Delphi configuration arrives at node 1 in Figure 8.12. It checks in to the Controller to see if there are any idle processors with which to split the choice point. When this check-in is performed at node 1, we will consider two cases:

(1) There are no idle processors waiting for work.

(2) There are two idle processors waiting for work.

In the first case the single processor continues to traverse the search space and reaches node 2. It performs another check-in at this node. Assume again that no idle processors are available. The single processor proceeds to the choice point labelled 3 and does a final check-in (no idle processors again). The processor then finishes executing the remainder of the search space. The total number of check-ins in this situation is three.

In the second case we assume that two processors are requesting work when the check-in occurs at node 1. The three processors split the work. The original processor (PE1) takes the left-hand branch. The other two processors (PE2 and PE3) explore the right-hand branch. These two processors both arrive at node 2 where no check-in is performed. Check-ins only occur when there is a processor working on its own within a subtree. At node 2, PE2 takes the left-hand branch with PE3 going to the right. PE2 then arrives at node 3 and performs the final check-in. The total number of check-ins in this case is two.

The results shown in Table 8.3 demonstrate the number of check-ins performed for each of the initialised check-in intervals. These results demonstrate the typical number of check-ins performed irrespective of the number of participating processors or the speed of the processors. The number of check-in communications is related to the initialised check-in interval with a random component for each individual run.

A comparison of the execution times for three N-queens problems are given in Table 8.4. The sources and queries used to obtain these results can be found in Appendix 8b. The execution times shown for Delphi are for a twenty processor configuration.

Problem	C-Prolog	Cosmic Prolog	Delphi (20 processors)
8-queens	411	175	42
9-queens	1994	841	127
10-queens	10245	4560	450

Table 8.4 N-Queens - Comparison of Execution Times

8.3 Other Problems

The N-queens problems are special because they do not require any control communications. This is because the search space is fairly balanced. The same is true for the contrived `ortest` program. These problems can be run with the check-in interval initialised to a very high number so that no check-ins ever occur. The same is not true for any of the other nondeterministic problems that were run on Delphi. These problems require a check-in interval that is less than the maximum depth of their search space and therefore require some communications to control the search. The sources, descriptions and queries used to test these nondeterministic programs are listed in Appendix 8b.

Problem	C-Prolog	Cosmic Prolog	Delphi (20 processors)
parser-2	38	38	11
parser-3	108	100	25
parser-4	390	354	73
adder	1192	805	154
pentominoes	19005	13474	2521
8-queens	411	175	42
9-queens	1994	841	127
10-queens	10245	4560	450

Table 8.5 Comparison of Execution Times

A comparison summary of the best Delphi results are shown in Table 8.5. In all cases the best results were obtained with a twenty processor configuration. Only with the 8-queens and 9-queens problems were the best results also obtained with fewer processors. The N-queens problems are in an unusual category. No control communications are needed so the check-in interval can be set to infinity. Most of the other problems show improved but not optimal results when no check-ins are performed. The parser-3 problem in Figure 8.13 demonstrates this. When the check-in interval is set to infinity some OR-parallelism is exploited. The execution time does decrease slightly as the number of processors goes from one to twenty. When the check-in interval is set to 20, better results are obtained. With all problems tested, excepting the N-queens, automatic partitioning alone is not enough. Without control communications, a twenty processor configuration is too small to exploit much of the OR-parallelism in these problems. To execute these problems efficiently, work must be reassigned to the idle processors. In all but the N-queens problems the reassign jobs strategy was used. The reassign jobs strategy was tried with various check-in intervals to obtain the results in Table 8.5.

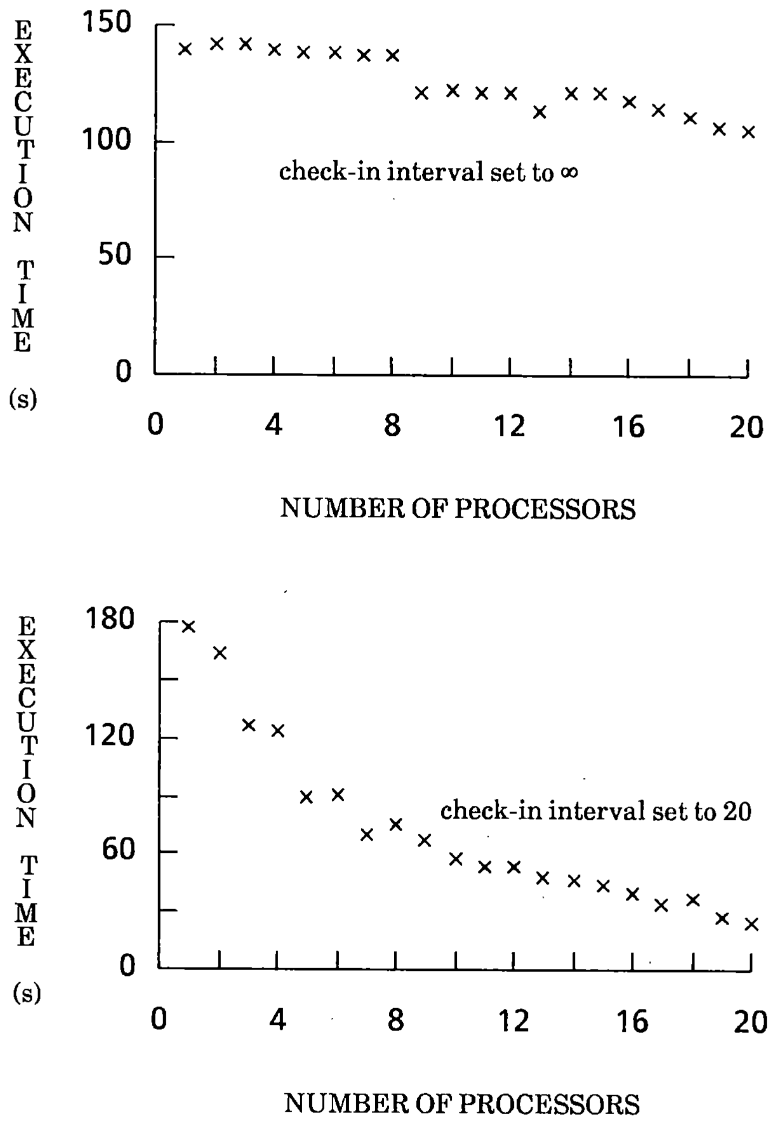


Figure 8.13 Parser-3 Problem - Two Check-in Intervals

Table 8.6 shows the check-in statistics for a twenty processor Delphi configuration running each of the problems. The first column is the initialised check-in interval. This parameter is sent to each of the configurations before the run begins. The only other initialisation parameter that is sent is the number of participating processors. The second column shows the number of check-ins actually performed for the given check-in interval. The third column shows the range of check-ins that occurred with configurations from one to twenty processors.

Problem	initialised check-in interval	number of check-ins performed	check-in range for all configurations
parser-2	10	158	158-177
parser-3	20	169	168-185
parser-4	40	144	144-153
adder	140	1921	1907-1978
pentominoes	40	20434	20279-20651
8-queens	∞	0	0
9-queens	∞	0	0
10-queens	∞	0	0

Table 8.6 Check-in Statistics

All of the problems have some communications associated with them. These communications can be split into four groups:

- initialisation
- logging messages
- answers
- control communications

Initialisation communications are a simple way to alter the parameters of a Delphi run. It is possible to have these parameters hard-wired into the Prolog processes or read from a local initialisation file. For convenience to the user initialisation parameters are given to the Controller process and this process automatically distributes them to the Prologs. In all of the strategies one initialisation communication takes place for each host machine. In the automatic partitioning and reassign jobs strategies the initialisation message consists of two integers. The first is the unique identification number for the Prolog process and the second is the number of initial Prolog processes configured for the run.

Logging messages occur as the Prolog systems are initialised on each host machine. In this case the messages are sent from the host machine to the Controller. The Controller also generates logging messages. This is usually the result of some compilation flag being set. An example of a logging message produced by the Controller is the entry in the global log file showing the total number of check-ins performed. Logging messages are also sent if an error occurs in any of the processes.

Problem	initialisation parameters	number of check-ins (control communications)	number of logging messages	number of answers	total number of communications
parser-2	20	158	23	5	206
parser-3	20	169	23	14	226
parser-4	20	144	23	42	229
adder	20	1921	23	16	1980
pentominoes	20	20434	23	8	20485
8-queens	20	0	23	92	135
9-queens	20	0	23	352	395
10-queens	20	0	23	724	767

Table 8.7 Communication Statistics for Twenty Processor Configurations

All answers to a query are collected in the global log file. This file resides on the same host machine on which the Controller is executed. Answer and logging messages are duplicated in a local log file on each of the host machines. This provides some redundancy so that if the network communications break down the answers will still be available. With error messages it may be impossible to transmit the message over the network. For this reason these messages are also duplicated in local logging files.

Control communications incorporate the sending and receiving of oracles. An oracle is sent to the Controller when a Prolog checks in. This oracle locates that Prolog within the search space. If there are idle processors waiting for work, the process which has checked in receives information containing the number of idle processes. When an idle process is awakened an oracle and some control information is sent to it. The control information consists of a pair of integers. One integer designates that Prolog's unique number in the group, the second is the total number of Prologs participating in the group. These messages are examples of control communications.

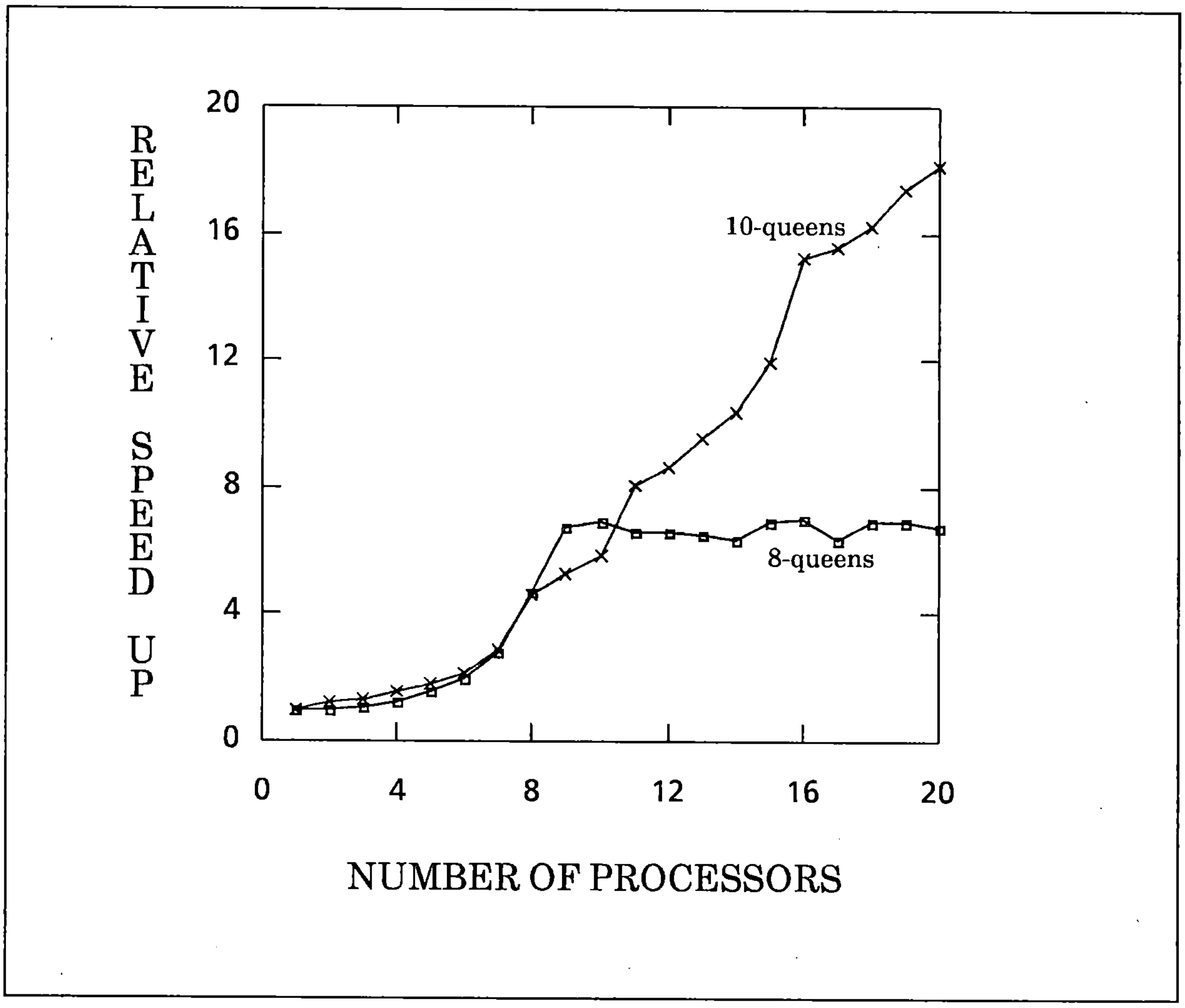


Figure 8.14 Relative Speed Up of Two N-Queens Problems on μVAXes

Table 8.7 is a summary of the communications for each problem. The number of check-ins is taken from Table 8.6. The number of *logging* messages is constant for each of the problems. Twenty of these are messages from the twenty host machines when they are initialised. Two of the logging messages are the start and stop messages for the execution-time clock. The final message is to report the total number of check-ins for the run. These comprise the twenty-three messages logged for each problem.

8.4 Relative Speed Up

Relative speed ups for the 8-queens and 10-queens problems running on the μVAXes are shown in Figure 8.14. A table showing the relative speed up for all of the problems is Table 8.8.

Problem	execution time on one processor	excution time on twenty processors	relative speed up
parser-2	58	11	5.27
parser-3	177	25	7.08
parser-4	461	73	6.32
adder	1814	154	11.78
pentominoes	26798	2521	10.63
8-queens	301	42	7.17
9-queens	1405	127	11.06
10-queens	8137	450	18.08

Table 8.8 Relative Speed Up

8.5 Faster Processors

Delphi was ported to a group of HP 9000 Series 350 workstations (also known as Bobcats). The 10-queens problem was used as a comparative benchmark. The results of the 10-queens problem running on the Bobcats is shown in Figure 8.15. For a single μVAX configuration the execution time of the 10-queens problem is 8137 seconds. On the Bobcats this same benchmark takes 3039 seconds, approximately 2.68 times faster. It would be expected that the N-queens problems would execute with a relative speed up as that shown in Figure 8.15. Since there are no extra control communications involved the relative speed up should be similar to that shown for the μVAXes in Figure 8.14. The interesting question is whether the performance holds up when control

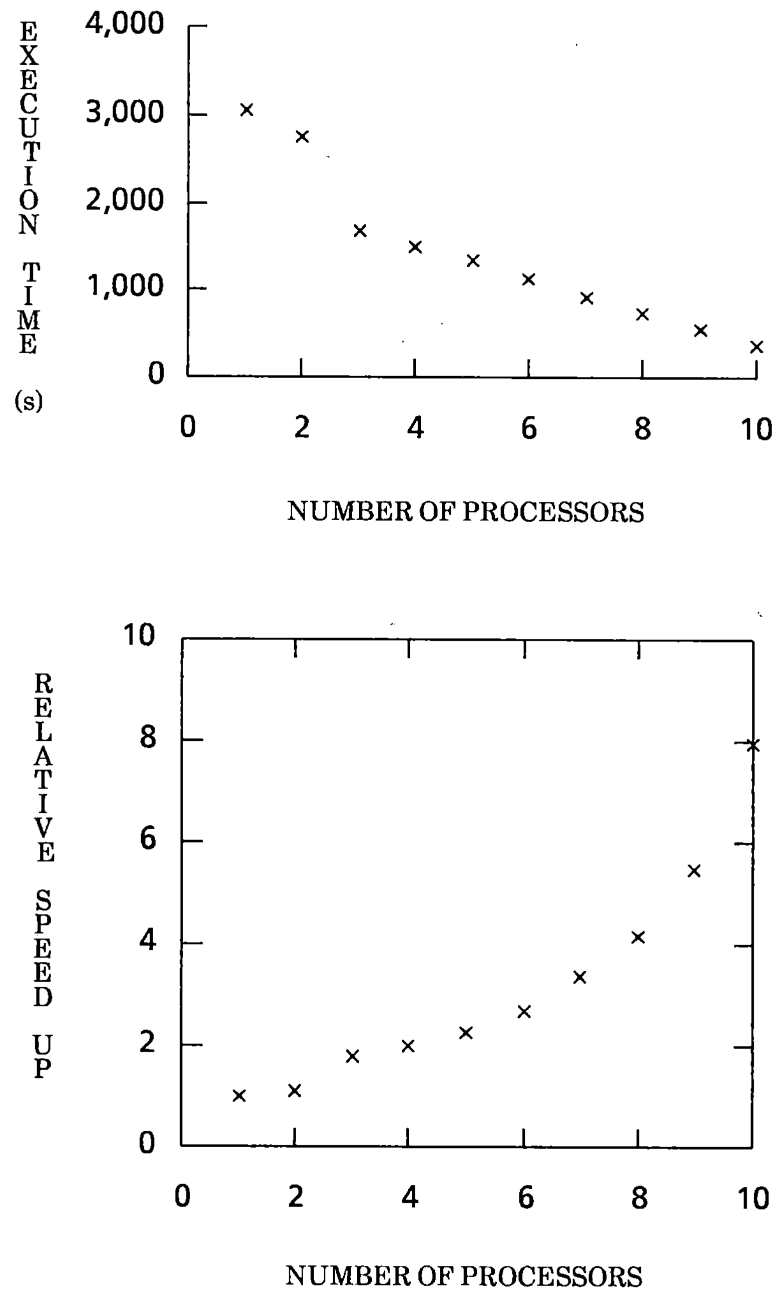


Figure 8.15 10-queens on the Bobcats

communications are involved. The network (Ethernet) being used is the same as that used for the µVAXes. Though the processor speed has been increased the speed of communication across the network remains the same.

A few of the problems were run on the Bobcats to ascertain whether the performance is hindered by the communications traffic. The parser-2 problem runs in seven seconds on a single processor configuration. This time could not be improved upon by the addition of more processors. In this case the execution time went up slightly (about three seconds) as the maximum number of processors (ten) was reached. The parser-3 problem was tested with a range of check-ins varying from ten to eighty. The best results were found when the check-in interval was initialised to thirty. This is ten higher than the results shown for the µVAXes in Table 8.6. This means that the communications are having an effect with the faster processors. The check-in interval has to be set a little higher so that not as many check-ins are performed. The execution time using ten processors was nineteen seconds.

The results using faster processors suggests the question of how to predict what the appropriate check-in value is when we know the relative speed of a machine. This question was not answered during the course of this research. To obtain a general solution to this problem a variety of processors with different speeds would need to be available for tests. One other possibility is to artificially reduce the processing power available to simulate slower processor speeds (for example by having a controlled interrupt overhead). Obtaining a solution to this problem would still not be very useful. A program would have to be tested with a large range of check-in intervals on some set of processors to obtain an initial optimal check-in interval. What is really wanted is a control strategy that is optimal for any class of problems running on processors of any speed. It is not known whether any control strategy will fit this requirement.

The adder problem was also run on the Bobcats with similar results to the parser-3 problem. The initialised check-in interval was set to 180. With ten processors the execution time was 131 seconds. Again the communications have required the check-in interval to be set higher than on the µVAXes. With the µVAXes the check-in interval was initialised to 140.

8.6 Deterministic Problems

Figure 8.16 shows the results for the matrix multiplication problem running the queries test20 (mm20 data) and test40 (mm40 data). With Cosmic Prolog the test20 query runs in 12 seconds. This is the same amount of time as for the one processor Delphi run. The execution time range from 12 to 18 seconds include the overheads due to setting up log files and sending the result over the network. The extra time also includes all of the calculations done each time the choice point is reached. The matrix multiplication problem in Appendix 8b does have an OR branching point for the predicate mmc. This is due to the fact that Delphi indexes the clauses only on the first argument. Every time this choice point is reached, calculations are done to see if it is time to check-in to the Controller. The range of from 12 to 18 seconds shows that this branching point is not reached very many times. With the larger mm40 problem this choice point is reached many more times as can be seen from the execution time.

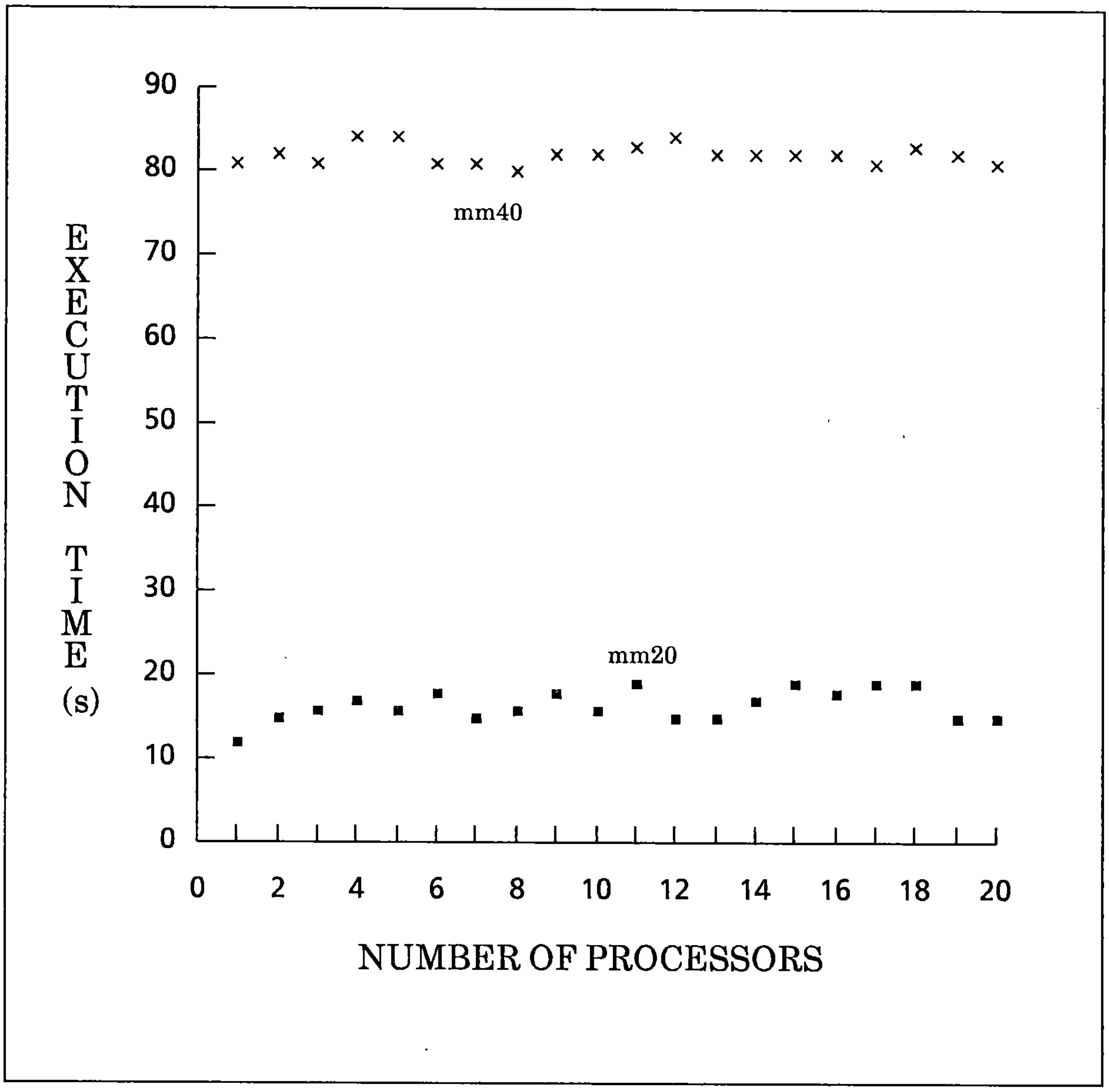


Figure 8.16 Two Matrix Multiplication Queries

Any *deterministic* program run on Delphi will show an increase in the execution time when compared with Cosmic Prolog. The deterministic problems which favour Delphi are those with no OR branches. With these problems we can expect to have an additional few seconds added on to the execution time of the sequential Prolog. The reason this number will be fairly constant is that during a deterministic problem there are no (or very few) OR branches encountered. It is only when an OR branch is reached that any extra computation is performed. Without any OR branches the Delphi machine behaves in a manner similar to Cosmic Prolog.

The potential additional overheads of running a deterministic problem (with no OR branches) on Delphi are described below:

- initialisations over the network

  Overheads involved with starting the Prologs on the various host machines over the network. This includes setting up local logging files for each of the Prolog systems. A global log file is set up on the machine where the Controller runs. The extra time depends on the load on each host machines and the status of the network.

- communication of the answer

  The overhead of sending the answer over the network to the global log file as compared to printing it out locally. For a deterministic problem at most one answer is received so this overhead is negligible.

- ensure no duplicate answers

  In a deterministic problem with no OR branches all Prolog systems will follow precisely the same path. With Delphi there is no possibility of more than one Prolog system reporting the same answer back to the Controller. With the automatic partitioning strategy each Prolog system always knows its own unique identifier and how many Prolog systems are currently exploring the same path. If there is more than one Prolog that reaches a solution then only the Prolog with unique identifier 1 sends the answer to the Controller.

The overheads involved with running a deterministic Prolog program on Delphi are proportional to the number of OR branches involved. If there are very few or no OR branches the overheads are small. Though Delphi cannot improve the performance of a deterministic program it also does not greatly hinder the performance. A comparison of the results for the two matrix multiplication problems is shown in Table 8.9.

Problem	C-Prolog	Cosmic Prolog	Delphi Range
mm20	29	12	12 - 18
mm40	198	67	80 - 84

Table 8.9 Matrix Multiplication - Comparison of Execution Times

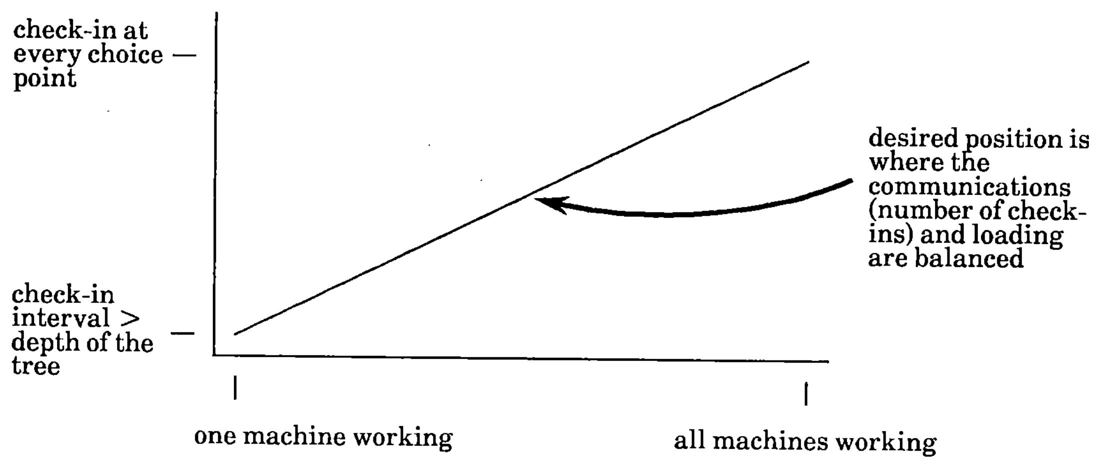
Chapter 9 Adaptive Control

Further control strategy developments are described in this chapter. Desirable features of some of the previous strategies are pointed out for use in the design of potentially more efficient and generalised control strategies. Some backtracking strategies together with their automatic partitioning mode have been modified by the addition of adaptive control behaviour. This behaviour provides a reduction in the total amount of communications by limiting the number of check-ins performed. A new class of control strategies with *active control* are described. In these strategies a reversal of the protocol for initiating communications occurs. The final control strategy described in this chapter involves making an estimate of the amount of work left to be done by each Path Processor. This estimate is then used to determine how often the Path Processor should communicate with the Controller. The work estimate provides a useful side effect of being able to give the user an estimate of the amount of time left for processing the problem. Both *active* and *passive control* techniques.are incorporated within this *work estimate* strategy.

9.1 Load Balancing Comparisons

With the non-backtracking strategies, load balancing among the host Path Processors was automatically adjusted. As soon as a Path Processor became idle it immediately was given the job of exploring the next oracle on the jobs queue. The Path Processors received a fairly equal number of oracles to explore, and the time each of them spent performing computation was roughly equivalent. With non-backtracking strategies, the time spent in communications to the Controller was also equally distributed though it was very high. Load balancing with the backtracking strategies has not been dealt with in such a successful manner. Many Path Processors may remain idle while others receive the bulk of the search tree to explore. This is not necessarily an unwanted property. If a low execution time is the goal and there are many host machines available to run Delphi processes, then there is no reason not to use them. Since Delphi runs on time sharing machines other users can be getting the benefit of the idle machines which Delphi initialises, but does not give much of the search space to (for exploration). Another reason to desire an unfair split of the work load is if communication costs across the network must be kept to a minimum. An automatic partitioning only strategy could be used to execute a program and avoid network communications at the expense of a potential unequal distribution of the work load.

In the reassign jobs strategy the check-in interval is one of the initial parameters given to each of the Path Processors (PPs). Setting this interval is used as a method of trying to statically adjust the load balancing versus communications costs. As the check-in interval increases towards the limit of the maximum depth of the search space, the PPs never check-in to enquire about any other PPs waiting for work. Communication costs decrease as this check-in interval increases, but the load balancing becomes very inefficient to the point where one or a few machines are assigned the majority of the tree to explore. Figure 9.1 demonstrates the situation. When the check-in interval is very high a few machines are doing most of the work. As the check-in interval is brought down to the minimum of one, all of the machines are working during most of the execution time.



check-in at every choice point
check-in interval > depth of the tree
desired position is where the communications (number of check-ins) and loading are balanced
one machine working
all machines working

Figure 9.1 Balanced Loading and Communications

In adjusting the check-in interval we are looking for a position (in Figure 9.1) on the line where the amount of communications is not hindering the performance of the program and as many machines as possible are in use throughout the computation. Without several trials of different initial check-in intervals, it would be difficult to find a balance between the amount of time spent performing communications as compared to the amount of time spent exploring the search space. Without this trial and error process, to properly set the check-in interval would require some advanced information about the problem to be solved and the shape of the search space.

If the problem is a binary tree such as the 8-queens problem then we would pick a very simple partitioning strategy. The low branching factor means that the partitioning strategy will not be too important so a simple splitting algorithm will probably be most effective. With the knowledge that the tree to be searched is binary, we would choose the partition right splitting algorithm. Since the maximum depth of the tree is around 70, we might pick an initial check in interval of around half this or 45. Better still, if we knew that the problem formed a fairly balanced search space, then we might set the check-in interval to ∞ so that no check-ins at all will be performed. Table 9.1 shows some examples of what parameters we might pick if information about the problem was available in advance.

TYPE OF SEARCH SPACE	CHOICE FOR STRATEGY
tree is deep but not wide	reassign jobs with a high initial check-in interval
tree has a high branching factor	reassign jobs with partition central splitting algorithm
tree is deterministic	branch by branch strategy (would utilise only one processor)
tree is biased to the right	reassign jobs with partition right splitting algorithm
tree is fairly balanced	automatic partitioning only

Table 9.1 Type of Control Strategy for a Known Search Space

If no information was available the first strategy to try is probably the reassign jobs strategy with a central partitioning algorithm. All of the available processors should be used with a check-in interval set to around forty. What would be much nicer to have is an automatic method for choosing a strategy, or changing the strategy or its parameters during execution. Automatically choosing an initial strategy or changing that strategy dynamically would involve some analysis of the program throughout execution. Automatically altering the parameters of the strategy as it is working, however, can easily be done to the existing control strategies. We will consider strategies which can automatically adjust their parameters in an attempt to balance the work load and minimise communications time. For some of these strategies additional storage such as a local queue needs to be maintained. This extra parameter of storage was originally described in the non-backtracking strategies. In addition to the communications costs, these strategies entailed overheads for the maintenance of job queues and for the memory they use.

Load balancing was not a consideration at all in the non-backtracking strategies. As the Path Processors become idle they pick the next job off of the jobs queue and are sent right back to work. If it is the case that there are no jobs in the jobs queue the Path Processor is put into a holding queue until a job becomes available. Usually this is a very short period of time since the working processors are sending oracles to the queue at every choice point. Only in the case of running a deterministic problem (with no backtracking), or a problem with only a minor amount of backtracking, will the load be shifted from all of the processors to just one or a few of them. Using only a few processors is exactly what you would want if the problem has very little OR-parallelism to exploit. The successful load balancing of the non-backtracking strategies is overshadowed by the time spent in performing communications. Because of the communications overheads, the non-backtracking strategies are not efficient enough for the exploration of large search spaces. It can be shown that the branch by branch strategy is at least as inefficient as the reassign jobs strategy with the Path Processors communicating at every choice point.

There is a similarity in the communications costs between the branch by branch strategy and the reassign jobs strategy when the check-in interval is set to one. In a comparison of these two strategies, we will look at the average communications costs, storage requirements, and execution overheads that the Controller witnesses. The conclusion to be reached is that the branch by branch strategy is at least as inefficient as the reassign jobs strategy with an initial check-in interval of one. Since this is the worst case set up for the reassign jobs strategy, setting the initial check-in interval to be any number greater than one will be more efficient than the branch by branch strategy.

In this comparison, we will consider the overheads that the Controller observes and not consider the point of view of the Path Processors running the Prolog. The PPs see a more complicated algorithm with the backtracking strategies than with any of the non-backtracking strategies. Even with the more complex backtracking algorithms, it appears that putting more of the computational burden onto the Path Processors rather than on the Controller leads to an overall more efficient system. One reason for the computation burden to be placed on the PPs and not the Controller is to avoid communication bottlenecks as the number of processors is increased. This is one reason why the branch by branch strategy was developed from the bit by bit strategies. Bit by bit strategies placed much more of the computation onto the Controller.

In the bit by bit strategies, the Controller had to generate and store oracles at the same time as it was receiving messages from the Path Processors. With the branch by branch strategies, the oracle generation process took place at the PPs' side. With the PPs communicating at each choice point, the number and frequency of the messages is high in either strategy. As the number of processors is increased it becomes increasingly more difficult for the Controller to keep up with answering these messages.

An implementation detail of the branch by branch strategy reduced some of the computation performed by the Controller and placed the burden onto the PPs instead. Complete oracles for all branches at a choice point (excepting the one that the PP will explore itself) are transmitted to the Controller and placed on the jobs queue. Originally, the number of branches and a prefix oracle were sent. The Controller could then recreate the proper oracles from this information and place them on the jobs queue. Even with this modification to reduce the Controller's execution overheads, it is constantly fighting to keep up with the Path Processors sending in jobs at every choice point.

We now compare the branch by branch strategy to the reassign jobs strategy with check-in interval set to one. We assume we are searching a tree with constant branching factor equal to B and depth equal to D.

COMMUNICATIONS

Reassign Jobs	Branch by Branch
For every choice point reached, for every processor, the path of where that processor is, gets sent to the Controller. The length is $\approx \frac{1}{2}$D.	For every choice point reached, for every processor, each unique path that the processor does not take is sent to the Controller. The length is $\approx \frac{1}{2}$D(B - 1).
Oracles sent to previously idle processors from Controller.	Oracles sent to previously idle processors from Controller.
Message sent back to the processor checking in containing the number of idle processors.	Receive acknowledgement.

STORAGE REQUIREMENTS

Reassign Jobs	Branch by Branch
No storage kept.	Current queue length plus B - 1 new jobs of average length $\frac{1}{2}$D.

EXECUTION OVERHEADS

Reassign Jobs	Branch by Branch
Check queue for idle processors.	Check queue for idle processors.
	Add oracles to the jobs queue.

Both of the strategies do some kind of communication at every choice point. In general, the branch by branch messages sent at each choice are longer than the reassign jobs messages. Each of the branches in a choice point (except for the leftmost, which has been chosen by the PP sending the message) is formed into a separate oracle before being sent to the Controller. The reassign jobs strategy sends exactly one oracle (the current path oracle) when a choice point is reached. The second communication message listed is equivalent for both strategies. Idle PPs are sent the oracle describing the path which they are to follow. The third communication message also has an equivalent length.

Storage requirements for the branch by branch strategy exceed the backtracking strategies since a queue of oracles which need to be followed must be maintained. Execution overheads additionally occur as this queue must be maintained. There are large numbers of oracles which have been discovered waiting to be explored by the small number of Path Processors. Storage for all of these oracles must be found and efficiently handled.

Communication times have been shown to have a significant effect upon the total execution time of a Prolog program. The execution time for the 8-queens problem configured for one PP and a check-in interval set to one is 8321 seconds. For one PP and the check-in interval set to two, the execution is 3970 seconds. For any number of PPs configured, the execution times with the check-in interval set to two is less than half the time when the interval is set to one. The communication overheads in the non-backtracking strategies would be at least as dominant a factor as the reassign jobs strategy with a check-in interval set to one. This is why the non-backtracking strategies were not implemented for a second time after the Amoeba-transactions-under-UNIX IPC mechanisms had been replaced.

9.2 Passive and Active Control

From the results shown in the previous chapter, a reduction in the number of communications is critical in the design of an efficient and generalised control strategy. So far, all of the control strategies shown have been given a specific interval at which communications occur. This interval was either a static parameter embedded in the control strategy, or as a parameter provided during the initialisation phase of Delphi execution. The non-backtracking strategies are forced to communicate at every choice point so this interval is embedded in the strategy. In both of these situations the control is termed passive control. Passive control is where the Path Processors check-in to the Controller at constant intervals throughout the execution time of the Prolog program. There is an alternative method where the number of communications is dictated dynamically at execution time and is not a static parameter. One method of achieving this is to modify the existing passive control techniques with adaptive behaviour.

In all previous strategies we have considered the communication requests progressing in a pattern from the Path Processors to the Controller. The Path Processor initiates a communication by sending a message to the Controller. The Controller responds with an acknowledgment or by sending the information that the Path Processor requested in the originating message (see Figure 9.2). With this situation, when a Path Processor sends a message to the Controller it must then be blocked until it receives a return message from the Controller. No more useful work is being done by the Path Processor until this reply arrives. A sample set of the possible initiation and reply communications for the reassign jobs strategy are given in Table 9.2.

On the Controller side, there is never any blocking done while waiting for a message to appear on any of its ports. A `select` system call [ULTRIX-32 Programmer's Manual: Sections 2,3,4, and 5 1987] is performed on all of the available communication ports to check for pending messages. This call is used in a non-blocking manner. If there is a message available on one or more of the ports the lowest numbered port is picked to be answered first. Successively higher file descriptors are then checked until all requests have been answered. The function of the Controller is to scan all of

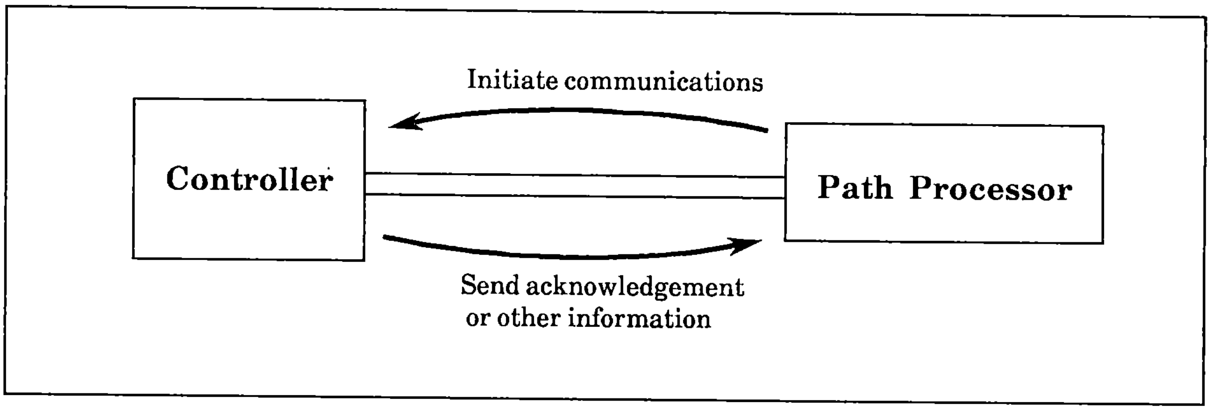


Figure 9.2 Communications are Initiated from the Path Processors

REQUEST FROM A PROLOG PROCESS	REPLY FROM CONTROLLER
get initial parameters	Initial U (unique identifier) Initial number of PP's (G value) Initial check-in interval
send answer	ACK
send message to be logged in the global log file	ACK
no more work to do (NEEDNEWJOB)	Oracle to follow New value for U New value for G

Table 9.2 Sample Requests and Responses for the Reassign Jobs Strategy

the ports for incoming messages, pick a port to answer, decode the request, and send a reply. Each Prolog process only has a single port so their communications protocol is more simple; send a message when necessary then block until a response is received.

There is no reason why the communications have to go in the predictable manner of Path Processor initialisation and Controller response. We can consider a generalisation of communication between the Controller and any of the Path Processors where either side can initiate a request. In this case, the protocol must be carefully defined and thoroughly debugged as all types of race conditions become possible. One of the reasons for wanting a more flexible facility is to allow the Controller to be able to "interrupt" any of the Path Processors. Allowing the Controller to interrupt any of the Path Processors would reverse the direction of the communication initiation procedure and create a control strategy using active control. Active control is part of an *on demand* strategy where the number of communications (check-ins performed) is equal to the number of times that the processors have become idle. With this strategy, check-ins only takes place when a processor reports that it has no work left to do and becomes idle. The Controller than finds an active Path Processor to interrupt and requests that PP to perform a check-in.

The interrupt sent to the PPs could be in the form of a signal which is passed and processed by an interrupt handler. It could also be a soft interrupt in the guise of a message sent to the Path Processor. Either type of interrupt would have to be acknowledged and held by the PP until it was able to respond to the request. When the PP was in a convenient position within the execution of the Prolog program (at a `setmax` instruction), then it could perform a check-in to the Controller for partitioning of its section of the search space. The reason for interest in the addition of an interrupt facility is to try and maintain a more equal work load on all of the processors. Only checking in on demand would additionally avoid the communications associated with the many failed check-ins. A check-in can be considered a failure is there were no idle processors to split the choice point with when the check-in occurred.

9.3 Limiting the Number of Check-ins

Limiting the number of check-ins performed is important since the communication overheads appear to be a dominant factor in the overall execution time of a Delphi run. Using the reassign jobs strategy on the 8-queens problems has clearly demonstrated this. When the check-in interval was set low the number of communications was high and the execution time of the problem increased. The N-queens problems, in fact, show very good results when no check-ins are performed at all. It is not the case that this sort of behaviour occurs with all problems. For most problems a trial and error process of altering the check-in intervals was used in an attempt to decrease the execution time. Altering the check-in interval was a static method of limiting the number of check-ins performed and therefore the communication overheads.

Adaptable methods for limiting the number of check-ins have been added to the existing backtracking control strategies. The major alteration to the passive control techniques is to inflict a penalty for checking in when there are no idle processors waiting for work. The penalty varies depending on the mechanism which caused the failed check-in to occur. In the reassign jobs strategy for example, if a Path Processors performs a check-in which fails (there were no idle PPs waiting for work) the penalty is to increase the check-in interval. This has the feature that Path Processors will eventually be performing check-ins at different check-in intervals. This is in contrast to the static method described with the reassign jobs strategy whereby all PPs check-in at the same interval throughout the processing.

Along with the concept of penalising the Path Processors for performing unnecessary check-ins, additional methods of deciding when to check-in have been implemented. With the reassign jobs strategy it was the exploration at particular depths of the search space which elicited a check-in by the Path Processors. The penalty for failed check-ins with this strategy will be to increase the depth interval. If there are Path Processors idle when the check-in is performed then the penalty is not imposed or a positive adaptation is performed. Table 9.3 shows three mechanisms for initially limiting the number of check-ins performed, and the type of penalty incurred if the check-in should fail.

MECHANISM FOR ELICITING A CHECK-IN	PENALTY INCURRED IF THE CHECK-IN FAILS
depth interval	increase the interval
branching factor	increase the threshold value containing the number of choice points (with that branching factor) which must be reached before a check-in is performed
time	increase the time

Table 9.3 Check-in Mechanisms and Penalties

The first mechanism is the one used by the reassign jobs strategy. If a check-in fails then the depth interval is increased with a reduction in the number of check-ins performed. Various means of increasing the check-in interval can be used. In the implemented version a constant number is added to the depth interval for each successive check-in failure. If the check-in is successful then a similar action is taken to decrease the check-in interval.

The second mechanism listed in Table 9.3 has not been used in any of the previous strategies. The check-ins are performed not in accordance to particular depth levels, but according to the branching factor of the choice points. Strategies which perform check-ins in relation to the number of branches at a choice point are called *branching factor* strategies. To limit where the check-ins will occur a static parameter is initialised to the minimal branching factor. Setting a minimal value for the branching factor means that check-ins will occur only at choice points which contain at least that number of branches. One variation on this strategy is to only check-in when choice points with the maximal branching factor (within the entire search space) have been reached. To perform this variation takes two steps. First the largest value of `setmax` for the entire tree has to be discovered. Then every time a choice point with this maximum number of branches is reached, a check-in to the Controller is performed.

Check-in Interval	2 branches	3 branches	4 branches	5 branches	43 branches
20	289	116	101	85	177
30	114	50	58	18	77
40	53	18	35	13	34
50	11	7	5	3	5
60	12	4	12	5	12
70	1	0	0	0	0
80	1	0	0	0	0
90	1	0	0	0	0
140	0	0	0	0	0

Table 9.4 Parser-4 Problem - Branching Factor when Check-ins are Performed

With the eight queens problem the branching factor at each check-in is always the same; it is two. This is ensured since each predicate set (clauses within the same predicate having the same type of first argument) of the N-queens problem contains either one clause or two clauses. Check-ins are not performed at the deterministic nodes, so the branching factor at each check-in during the N-queens problem must be equal to two. In other problems this is not the case; the number of branches at the choice points exhibit a range of values.

While implementing the branching factor strategies, the reassign jobs strategy was examined to determine the distribution of branching factor values. When a Path Processor performed a check-in, the value of the setmax instruction was noted. An example of this experiment using the parser-4 problem is shown in Table 9.4. The check-in intervals range from 20 to 140, with the maximum depth of the tree being 131 levels.

The parser-4 problem has choice points containing five different branching factors; two branches, three branches, four branches, five branches and forty-three branches. For all of the tested check-in intervals, the choice points with two branches had more check-ins performed at them then any of the others. The minimal number of check-ins occurred in most cases at the choice points with five branches.

2 branches	3 branches	4 branches	5 branches	43 branches
11008	8990	3751	2315	8237

Table 9.5 Parser-4 Problem - Number of Nodes with each Branching Factor

A branching factor control strategy was implemented which only performed check-ins when the maximum branching factor in the tree was reached. The parser-4 problem was first attempted with check-ins only performed when the branching factor was equal to forty-three. The execution times using this strategy turned out to be considerably longer than the reassign jobs strategy. The reason for this is that in the entire search space, there are a large number of nodes with a branching factor equal to forty-three. For the entire search space the number of nodes with each of the number of branches is given in Table 9.5.

If check-ins were performed at every node in the parser-4 problem containing forty-three branches then the number of check-ins would be equal to 8237. This is much higher than the number of check-ins performed with the reassign jobs strategy. The number of seconds it takes to execute the parser-4 problem with twenty Path Processors is seventy-three. Performing 8237 check-ins in this time period would give a check-in rate of around 113 check-ins per second. This rate is much higher than any of those which produced the results shown in the previous chapter. The check-in rates for the Prolog problems in the previous chapter is given in Table 9.6.

To limit the number of check-ins performed with the branching factor strategies, a simple adaptation is added to the control strategy. A count is maintained of the number of nodes where

Problem	Execution Time	Number of Check-ins Performed	Check-in Rate
parser-2	11	158	14.4
parser-3	25	169	6.8
parser-4	73	144	2.0
adder	154	1921	12.5
pentominoes	2521	20434	8.1
8-queens	42	0	0
9-queens	127	0	0
10-queens	450	0	0

Table 9.6 Check-in Rates

check-ins would occur. If check-ins occur only when the maximal branching factor is reached for the parser-4 problem, then this count will contain the number of nodes reached which have forty-three branches. In addition to this count, a threshold value is maintained. Only when the count reaches the threshold is a check-in performed. Check-ins will only be performed after a certain number of choice points with the maximum branching factor have been reached. Penalties for failed check-ins and bonuses for successful ones easily can be added by increasing or decreasing the threshold value.

The final mechanism listed in Table 9.3 for controlling the number of check-ins performed is to use a timer. An `alarm` library routine [ULTRIX-32 Programmer's Manual: Sections 2,3,4, and 5 1987] can be used to count the number of seconds before the signal `SIGALRM` is sent to the process which set the alarm. The sending and receiving of signals creates the need for special code which may be specific to the machinery used. A technical note on using signals and some portability issues are discussed in Appendix 9a. The code for a simple program which uses `alarm` is shown in Figure 9.3.

```
#include <stdio.h>
#include <signal.h>

int sig_alarm();

#define         TRUE    1
#define         FALSE   0

typedef int Boolean;
Boolean AlarmFlag = FALSE;


main()
{
    signal(SIGALRM, sig_alarm);
    for(;;){
        alarm(10);
        while (!AlarmFlag);
        printf("\nALARM\n");
        AlarmFlag = FALSE;
    }
}

int sig_alarm()
{
    AlarmFlag = TRUE;
}
```

Figure 9.3 Using the Alarm Signal

Executing the code in Figure 9.3 results in the word ALARM being printed every ten seconds. The routine `sig_alarm` is the signal handler which catches the signal and sets a flag. The flag can then be polled at a later time when it is convenient within the code. If a timer mechanism were used to limit the number of check-ins performed, then the flag set by the alarm signal would be inspected when a choice point was reached. This *timer strategy* could be used on its own or in conjunction with any of the previously described mechanisms. For example, the following conditional statement could be formed to limit the number of check-ins:

```
IF    (at proper depth level AND branching factor is maximal AND alarm flag is set)
THEN    perform a check-in to the Controller
```

It is very easy to add adaptive behaviour with a timer strategy. The number of seconds set in the `alarm` routine can be decreased for successful check-ins and increased when a check-in fails. Another use of the timer strategy is to limit the number of check-ins performed in relation to the amount of execution time taken by the Path Processor. The total number of seconds could be estimated by counting all of the values used to set the alarm. The number of check-ins could then be increased or decreased as the execution time of the problem exceeded particular values.

9.4 Active Control Strategies

All of the methods demonstrated for limiting the number of check-ins have involved passive control. Even with the adaptive techniques for limiting the number of check-ins performed, it is possible for there to be no idle PP waiting for work when the check-in does occur. To amend this situation active control strategies are introduced. With an active control strategy it is the Controller which initiates communications with the Path Processors and tells them to perform check-ins.

The first control strategy described which uses active control is the on demand strategy. When a Path Processor becomes idle, the Controller sends a signal to one of the active PPs telling that PP to perform a check-in. When the check-in is performed by the interrupted PP it will always be successful. Implementation of this strategy is very similar to the timer strategy in that it involves a signal being sent and a signal handler to receive it. The major difference is that now instead of only one process being involved, the signal is sent by the Controller over the network to one or more of the Path Processors.

Implementing an on demand strategy requires the use of a signal which can be sent via a socket connection. One such signal is `SIGURG` which tells the receiving process that an urgent condition exists on the socket. Using this signal involves the sending of an *out of band data* message which is just a special flag on the `send` call. Two things occur when an out of band data message is sent:

- The signal `SIGURG` is sent to the process.
- The single byte which is the out of band data message is placed in the data stream.

The message is used to mark a place in the data stream so that the position of where the out of band data was sent can be used. This is often used to mark the position within a buffer where data is to be flushed up to. See Appendix 9b for a discussion of out of band data and an example of using it.

For implementation of the on demand strategy the data byte was sent into the stream but not used. A handler such as the one shown in Figure 9.3 receives the out of band signal and sets a flag. Just as on the timer example, the flag can then be checked and an action taken at a convenient spot in the code. The first strategy implemented using this facility involved sending a signal to an active process whenever an idle process reported in to the Controller. The Path Processors were 'interrupted' by this signal to tell them to perform a check-in. When the check-in is performed it will be successful since the Controller can keep track of when PPs are idle or active. An extra data structure was added to keep track of Path Processors which had been sent an interrupt and those which had not. In this way more than one PP could be interrupted at a time and the Path Processors in the idle queue could be distributed among the active processes.

The on demand strategy with an active PP being interrupted as soon as another PP becomes idle, turned out to be unsuccessful as compared to the passive control strategies. Path Processors become idle very often during the execution of a Prolog program. The extra computational burden placed on the Controller in performing interrupts to the PPs takes time away from the large number of PPs which are reporting in as idle. A parameter was added to the strategy so that an interrupt was sent only when a particular number of Path Processors had become idle. Still this did not improve the on demand strategy to the point where the execution times were competitive with those of the passive control strategies.

A combination strategy using the interrupt facility and the alarm signal was attempted. When the alarm was located on the Path Processors side, it was used to signal how often to check-in to the Controller. With the alarm placed at the Controller side, it can be used to control the out of band data messages being sent to the Path Processors. Again this active control strategy was not as successful as the ones using passive control. An adaptive measure could be used to limit the number of out of band data messages being sent. A penalty would be incurred if there were idle PPs blocked when the interrupt was sent by the Controller. The penalty would be to either increase the number of idle PPs in the queue before sending an interrupt or to increase the time before the next out of band data message is sent.

The passive control adaptation mechanisms have been implemented but not tested as widely as the automatic partitioning and reassign jobs strategies. These adaptations were implemented in the attempt to discover a general control strategy for any Prolog program. It is not certain whether these adaptations will generalise a particular strategy so that all problems will run successfully using it. In implementing the active control strategies it was found that too much time is spent in sending interrupts to the PPs. When there is only a single Controller it appears that passive control techniques are more efficient.

9.5 Strategies using Work Estimates

Many of the desirable features of the non-backtracking and backtracking strategies are brought together in the work estimate strategy. An additional feature in this strategy is the ability to give an estimate to the user of how much time is needed for executing the Prolog program. This is a desirable feature for all of the long running Prolog programs which are executed on Delphi. The following set of features are goals of the work estimate strategy:

- Keep as many of the Path Processors working as often as possible.
- Perform load balancing automatically.
- Use minimal communications.
- Maintain some measure of how much work there is to left to do.

Keeping the Path Processors busy is an easy task as long as the computation time spent on "useless" work is overlooked. Useless work falls into two categories: communication overheads and exploring portions of the search space which are too small. Spending time communicating with the Controller is useless work if the communications are performed too often. It is also useless to perform a check-in when there are no idle PPs. If the communications are sent in close succession, the Controller will not have time to deal with all of the requests and the PPs will be blocked waiting on a response. If there was no idle Path Processors at check-in time, the communication was wasted since the same PP will have to continue exploration on its own.

The exploration of very small portions of the search space is also useless work. An example is when an oracle which is rather long is given to a Path Processor to follow. A great deal of computation takes place in following the oracle to the point at which a unique portion of the search space will be explored. After the following phase is completed, the proper branch at the choice point is chosen and this path immediately fails. It would have been more efficient for the processor which sent the original oracle to have explored this branch by itself and not split the choice point.

If we do not care about this useless work occurring, we can easily maintain a queue of jobs that need to be done just as in the non-backtracking strategies. Maintaining this queue will ensure that there is always work available for a PP to do if and when it becomes idle. As an enhancement to avoid placing all of the burden for maintaining this queue onto the Controller, a limited number of oracles will be held at any time. In the non-backtracking strategies, the Controller must accept any oracle received from the PPs and place them on the queue. An oracle which is missed means that the tree would not be searched exhaustively and solutions may be lost. Maintaining a jobs queue with all of the paths which have so far been discovered but not explored wastes a great deal of space and time. To lessen the burden on the Controller a limit is placed on the number of jobs which can be held. An example of a limited queue is to maintain the number of jobs in the queue to satisfy the expression:

$$\tfrac{1}{2}\text{Number of Path Processors} \leq \text{Number of Jobs in Queue} \geq \text{Number of Path Processors}$$

This is not a very large number of jobs (at least for the Delphi configurations that we have so far considered) so the queue can be kept small and easily maintained. This expression is also an estimate of the number of processors which may become idle at the same time. If this number is exceeded then some of the PPs will have to go onto an idle queue until more work becomes available. If all of the Path Processors become idle at once then the problem has been completed. If half of the PPs become idle, there must still be portions of the search space to explore since half of the processors are still working.

Maintaining this jobs queue involves some method for acquiring new work if the number of jobs on the queue falls below a critical value. One of the objectives is to perform this job acquiring function automatically without any parameters supplied by the user (as in the reassign jobs strategy). Check-ins would only be performed if necessary, where necessary means that there are slots in the jobs queue to fill or worse, idle PPs waiting in a queue for work. If a Path Processor must be interrupted so that a check-in will occur, the PP with the largest work estimate will be chosen. This is an estimate of how much work each Path Processor thinks it has left to do at any time. Various algorithms can be described which give an estimate of the amount of work remaining in a partition. One possibility is the summation of all branches which have been allocated to a PP to explore. For a single PP exploring the search space on its own, this value will be the summation of all `setmax` values (branches at a choice point) which have been reached so far. Figure 9.4 is an example tree with summational estimates shown for the four internal nodes.

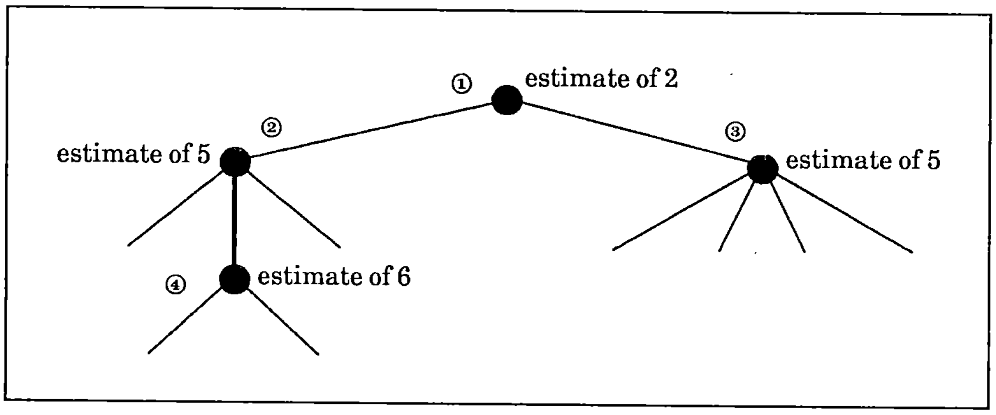


Figure 9.4 Summational Estimates

The estimates in Figure 9.4 are just a count of the number of branches that the processor knows exist but have not been explored. We assume that a single processor is exploring the search space on its own and that no limited choice points are created. On backtracking the count is reduced by one branch. When a choice point is reached the count is increased by the `setmax` value. This is a very simple estimate which gives information only about work that the processor knows to exist. No attempt is made to estimate how much work exists within the entire search space or portion of the search space that the PP has been allocated. This estimate can be used by the Controller in various ways. If the work estimates are maintained for each of the Path Processors, load balancing can be dynamically assessed by the Controller. Totalling all of the PP estimates will provide an estimate of how much work is know about in the search space as a whole. If a Path Processor needs to be interrupted to provide a job for the jobs queue, then the PP with the largest estimate can have its work load split.

The summational estimates in Figure 9.4 are not real estimates as they count known branches only. It could be made into an estimate by adding on an additional amount at each choice point. For example, if at each choice point the number of branches was doubled and then added onto the previous value, this would be a true estimate. For the tree in Figure 9.4, an estimate of four would

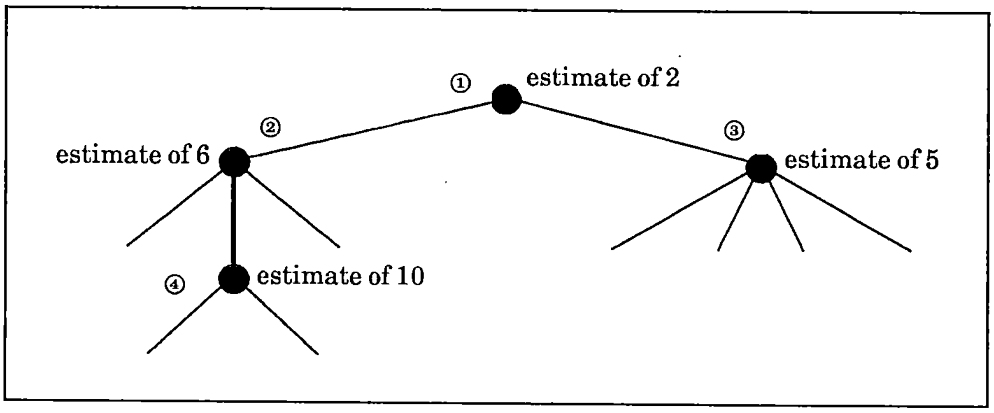


Figure 9.5 Multiplicative Estimates

occur at node one, an estimate of ten at node two, an estimate of thirteen at node four and an estimate of sixteen at node number three. This again assumes that on backtracking one branch is subtracted from the estimate.

A more useful estimate for a tree is shown in Figure 9.5. A multiplicative estimate multiplies the current estimate by the number of branches which have been allocated to that PP for exploration. This attempts to give a work estimate of the entire portion of the search space which has been partitioned and assigned to a Path Processor. For the estimates shown in Figure 9.5 we have again made two assumptions for simplicity. The first is that there is only one Path Processor which explores the entire tree. No limited choice points are created during the exploration. The second assumption is that on backtracking only a single branch is deleted from the estimate.

Check-in intervals perform a different job than these of work estimates. They tell the Controller when the backtracking level is outside of a certain range. The expression used to calculate if the current level is one at which a check-in should be performed is:

absolute value (current choice point − last choice point where a check-in was performed)
must be $\geq$ initial check-in interval

Using this expression to decide when a check-in is performed may not always show the places where a large amount of exploration is left to do. Quite a lot of backtracking could be going on by a PP, but until the required threshold value (as given by the initial check-in interval) is reached no check-ins will be performed. Communication overheads are attempted to be controlled by altering the initial check-in interval. The theory is that Path Processors doing a great deal of backtracking will reach the threshold many more times. These PPs must therefore have a lot of work to do and so they check-in more frequently to have their work partitioned. The work estimates can be more exact in providing a quantitative picture of how much work a PP has left to do. Unlike check-in intervals, the estimate is not necessarily related to the amount of backtracking which has been performed by a PP nor does it provide a means to control the amount of communications that will be performed.

Change in depth information is what the initial check-in interval is set to convey. If the check-in interval is initially given as N, then only a change of N levels in the tree will trigger a check-in to the Controller. A great deal of processing time could be spent backtracking within a shallow area and the Controller would not be aware that there still was work left to be done in the tree. An example of such a tree such can be seen in Figure 9.6. The filled triangles indicates an arbitrary number of branches at that choice point. Figure 9.6 shows a search space with a large branching factor at a few of the choice points and a shallow depth. If the initial check-in interval is not less than the maximum depth of the tree (in Figure 9.6 this depth would be less than five) no check-ins will ever be performed. Clearly there are places within this tree where it would be advantageous to perform a check-in. Though the check-in interval mechanism does not work properly on trees of this shape, the work estimates do take into account nodes at the same level which have a high branching factor.

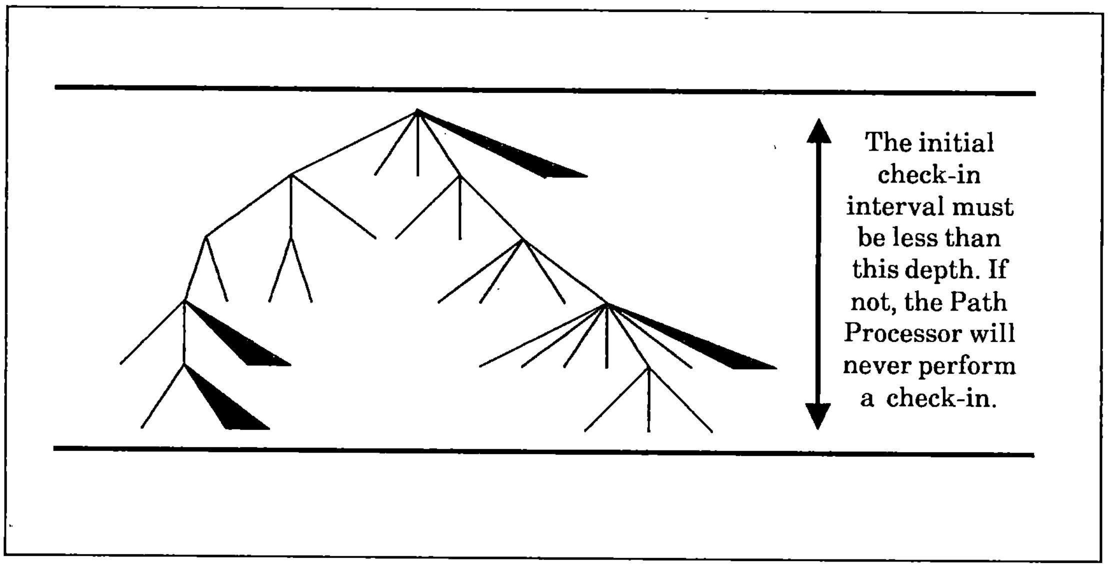

The initial check-in interval must be less than this depth. If not, the Path Processor will never perform a check-in.

Figure 9.6 Change of Depth Information

An algorithm using work estimates would begin execution in a similar manner to the reassign jobs strategy. Each Path Processor is given an initial unique identifier (U) and the total number of PPs beginning execution at the root of the search space. The tree is then automatically partitioned until each PP is exploring a unique section of the original search space. After this mode has completed various strategies using the work estimates can begin execution. Three such strategies are described which attempt to keep the Path Processors working:

- Use the interrupt facility as in the on demand strategy.
- Add a small queue of jobs.
- Combine passive control with the active interrupt strategy.

An extra amount of complexity is added to these strategies in the time required to maintain a sorted list of host machines and their most recent estimates. This data structure is maintained by the Controller in addition to the idle processors queue. With only a few host machines the estimates list could be kept as an unsorted array and exhaustively searched to find the PP with the highest estimate. If many PPs are initialised then this data structure must be kept sorted.

The use of estimates in a strategy such as the on demand strategy is so that a PP with a large amount of work remaining to be done can be targeted for interruption. To provide this facility only a small number of changes need to be made to the on demand strategy. The first modification is that when a Path Processor performs a check-in, not only is the oracle contained in its current path sent to the Controller, but also an estimate of the amount of work left to do. When a PP begins work on a new section of the search space its estimate will have to be reinitialised. The Controller could alternate between interrupting a PP which has a positive estimate to those which have just been given new work to do. In this way, all of the Path Processors have the potential of being interrupted. Path Processors are only interrupted when another PP has become idle just as in the on demand strategy.

The difference between the on demand strategy and using the work estimates in conjunction with the on demand strategy are minor. Work estimates are only sent when a PP performs a check-in, so no extra communications take place. The major differences are that an estimates list needs to be maintained, and the PP chosen to be interrupted is not necessarily the first active PP on the list (as in the on demand strategy). Maintaining the estimates list is not very different from the active/idle processors list which was used in the on demand strategy.

The following code is what each Path Processor will perform on reaching a choice point. The interrupt flag is set when the Controller sends out of band data to tell the Path Processor to perform a check-in.

```
When a choice point is reached DO
      check Interrupt Flag
      IF  Interrupt Flag is set
          send oracle and new estimate to Controller
          create a limited choice point
      ENDIF
ENDDO
```

This strategy will keep the Path Processors working, but will also burden the Controller with having to spend a great deal of time interrupting PPs. In an attempt to lessen this burden, a small jobs queue is maintained by the Controller. Instead of interrupting a Path Processor because another PP has become idle, the interrupts are used mainly to keep the jobs queue full. When the number of jobs in the queue falls below some particular critical level, then the Path Processors with the highest estimate (or one that has just been given new work to do) is interrupted. Since both an estimate and an oracle are sent when the PP performs a check-in, the oracle can be placed on the jobs queue and that PP's estimate updated after the single communication. The reason for maintaining the jobs queue is to quickly give the idle PPs new work to do by taking one of the oracles off the queue. This is faster than having the idle PP wait for an interrupt to be sent from the Controller and an active PP to check-in. The interrupts can be done at the expense of the Controller and not make the idle PPs wait in the jobs queue for work to become available.

The problem with both of the previous strategies is that the estimate value will not be very accurate. Estimates are only provided to the Controller when the Controller interrupts a Path Processor. The final work estimate strategy to be described uses passive control techniques to keep the estimate value up to date.

The reassign jobs strategy used a check-in interval to control the amount of communications from the PPs to the Controller. The Controller was not burdened with having to interrupt active Path Processors when a PP became idle. The idle PP had to wait until a check-in was performed before it could continue exploration of the search tree. This final strategy uses this passive control technique both to keep the estimates list up to date and to keep the jobs queue filled. Instead of using the initial check-in interval throughout the strategy, the adaptation methods described earlier in this chapter will be used.

When a Path Processor checks in to the Controller (with an updated work estimate and an oracle) it will be penalised if there are no idle PPs or any jobs needed to fill the jobs queue. If the check-in is successful then the check-in interval can be reduced for that PP to allow it to check-in more frequently. The amount of change in the check-in interval can reflect the importance of the successful check-in. If the check-in added one of the final jobs onto the jobs queue, then its interval would only be changed slightly. If however the check-in was successful because an idle Path Processor was found waiting for work, then the interval will be reduced significantly. This feedback is no more effort for the Controller than telling the PP which has performed the check-in the number of idle PPs in the queue. It is the Path Processor which updates its own check-in interval and leaving the Controller to ensure that work is always available for any PP which becomes idle.

If there are idle Path Processors waiting for work then the jobs queue must be empty. This is a serious situation since it may take time away from the Controller handling incoming idle PP requests. The Controller has two options in acquiring work for the idle Path Processor:

- Passively wait until an active PP performs a check-in.
- Interrupt the PP with the highest estimate to demand that a check-in be performed.

If the Controller waits for an active PP to check-in then it is possible that another PP will become idle in this time. If the Controller interrupts an active PP then it has taken time away from potential incoming idle PPs. These PPs will be blocked until the Controller finishes with the sending of an interrupt (out of band data) to an active process. In either case a check-in needs to be performed. When the check-in is performed, it would be useful for that check-in not only to provide a job for the idle PP requesting new work, but also for any additional PPs which may soon become idle. This is very easy to do by assigning *phantom Path Processors* to explore some portion of the work which is sent when the check-in is performed.

When a Path Processor performs a check-in, it receives the number of idle PPs on the jobs queue with which to split the work load. The Controller can "lie" to the Path Processor which has checked in by returning a number which is larger than the actual number of idle PPs. The Controller can then keep some of the work for itself, and put this work in the jobs queue. The group of PPs which participate in splitting this work, assume that there is an extra phantom Path Processor working as part of their group. The PPs will therefore leave some section of the search space untouched. When the next PP becomes idle, it will be given the remaining work to be done (the oracle which the Controller placed on the queue) and its unique identifier (U) and be able to finish the work left by the phantom PP.

Using these techniques, the Controller can attempt to have work immediately available for any PP which becomes idle. If the jobs queue runs out of work then new work can be acquired by assigning one or more phantom PPs to the group and keeping these jobs on the queue. If the feedback from passively performing check-ins is successful then the Path Processors may never need to be actively interrupted by the Controller.

Chapter 10 Delphi Tools

10.1 Search Space Analysis

Oracle instructions are contained within the intermediate code of any Prolog program run on the Delphi machine. The `setmax` oracle instruction has as its parameter the number of possible branches at each choice point. From this information it is very easy to create a data file containing a description of all of the paths searched while executing a Prolog program. This file is created by running Delphi on a single processor with some compilation flags set to print out the value of the `setmax` operand. A single processor Delphi searches the tree in a depth-first and left to right manner, so the data file created is in a depth-first format. This file is then analysed to give information on the path lengths, number of branches of each length, and other data which can be easily derived from a depth-first representation of the search space. To create a graphical representation of the tree, it would be more useful to have a breadth-first representation of the search space. For this reason, given enough disk space and time, the depth-first file can be transformed into a breadth-first file and then displayed in a graphical format.

Programs have been developed to analyse the search space using C , Modula-2 and many of the UNIX facilities such as `awk` and `sort`. The goal is to display the search space in a way that would be helpful to the user. One of the Modula-2 programs uses the breadth-first file and produces PostScript output. If the tree is very small it can be printed on a few sheets of A4 paper. Unfortunately, even something as tiny as the 4-queens problem is too large to be properly displayed by the current implementation. A second representation of the search space using a simple ASCII character format has also been developed. Figure 10.1 shows a simple example of the output from the tree analysis program.

The tree file is first analysed in a depth-first manner displaying information such as the number of answers found and at what depth they were found. The number of branches of each length is shown along with the mean and mode of the branch lengths. A breadth-first data file is then created from the original depth-first data file. For some of the larger problems, it may take too much time or space to create this file, so the analysis is only done for the depth-first file. The information given in the breadth-first analysis is slightly redundant, as the number of answers is again calculated. The new information generated by the breadth-first analysis is used to create a graphical representation of the search space.

The total number of nodes is given by the breadth-first analysis along with a breakdown of their types. Internal nodes can be divided into deterministic nodes (the `setmax` parameter is equal to one) and nondeterministic nodes (the `setmax` parameter is greater than one). In the example

```
              DEPTHANALYSIS

Number of answers = 1
Number of branches = 4, Maximum branch length = 3, Minimum branch length = 1
Number of branches of length 1 = 1
Number of branches of length 2 = 1
Number of branches of length 3 = 2
Average branch length = 2
Most common length = 3
Number of answers at depth 3 = 1

              BREADTHANALYSIS

Maximum branching factor = 2
Total number of nodes = 7: internal nodes = 3
     non-deterministic = 3, deterministic = 0
Number of leaves = 4: failed leaves = 3, answers = 1
Maximum width (number of nodes) on a level = 2 on level 1


                                  2____0
                                 F_2___1
                                 2_F___2
                                 A_F___3
```

Figure 10.1 Tree Analysis Output

shown in Figure 10.1 and in Appendix 10a, the number of deterministic nodes is always 0. An optional flag can be turned on when creating the tree data file to prohibit the display of any deterministic nodes. This option was used in all of the tree analyses given in this appendix. The reasons for providing this option are:

space considerations	If the deterministic nodes were added to the depth-first data file, these files would be much larger. The breadth-first analysis could not be performed on the larger data files. Any graphical representations would span over many pages.
representation	Delphi is an OR-parallel Prolog system, so it is reasonable not to display the deterministic nodes. The amount of parallelism which can be exploited is demonstrated by displaying only the nodes which have more than one branch.

An attempt is made to give a pictorial summary of the search space if the breadth-first analysis succeeds. If the tree is small enough to render on a single A4 sheet, it is represented as an ASCII file (see Figure 10.1). Numerals represent the branching factors of the internal nodes while the letters represent the leaf nodes. An 'F' (Fail) represents a leaf node that has been reached with no solution. It is a failed path. The 'A' (Answer) represents a leaf node with a solution. This is a successful path and an answer has been found. Each level of the tree is displayed to the right of the nodes at that level. With this representation, it is easy to see where the answers are located in a search space, and distinguish the widest and longest portions of the tree. An example of using the ASCII representation to construct a line drawn diagram is shown in Figure 10.2. The PostScript output produces trees similar to the graphical representation shown in Figure 10.2. Other examples of data file analysis and ASCII representations are in Appendix 10a.

```
   2________0

F__2________1

2__F________2

A__F________3
```

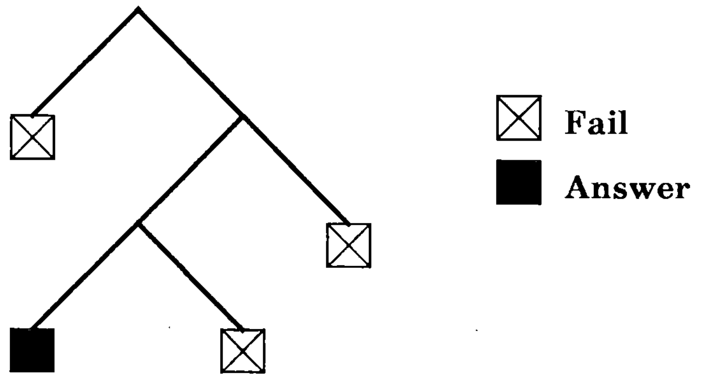


Figure 10.2 ASCII and Graphical Representations

10.2 External Process Management System

This system was originally designed for use with the Amoeba-transactions-under-UNIX implementation of the Delphi machine. It has been developed into a completely independent tool (independent of the Delphi machine) for managing a distributed system. The Process Management System (PMS) handles aspects of remote distribution from password management to checking on process status over the network. It is by far the largest and most general tool that was motivated by the Delphi research.

The goal of the PMS is to provide a simple and powerful user interface for process control over multiple host machines. Three types of processes are established by the PMS:

command process	This is the top level process which initiates and controls the other two types of process.
error server	Any problems with the system are reported to this process.
daemons	A daemon process spawns and watches all of the requested user processes on a particular host machine. There is one daemon running for each host machine.

The command process is a menu driven program initiated by the user. There are two groups of processes which are manipulated by this command process; the PMS processes, and the user process. Figure 10.3 shows the top level menu which is entered when the command process is executed.

Top Level Commands

```
    d       remotely distribute user programs using the rdist command
    h       print this menu
    k       kill remotely executing process(es)
    l       list remotely executing processes
    r       remotely execute a program on a given host(s)
    s       get status of processes on a given host(s)
    t       transfer files to a given host(s)
    u       perform an update on the process list

Commands that follow are used for initialising and checking the
status of the Process Management System.

    A       print a list of Amoeba Daemons known to this session
    B       implement shutdown procedure for the system -
                  this kills all daemons and all spawned user processes
    E       start Amoeba Daemons executing on host(s)
    F       copy/update Process Management System files on host(s)
    H       update hosts file
    Q       exit the Process Management control program (this process)
    R       execute restart procedure
    S       find status of Ameoba Daemons on host(s)
    T       terminate Amoeba Daemon on host(s)
    U       enquire about Amoeba Daemons from the Errorserver
    W       get errormessage about Amoeba Daemon from the Errorserver
    X       get the Errorserver started on a given host
    Y       find out if the Errorserver is up
    Z       tell the Errorserver to exit
```

Figure 10.3 PMS Top Level Menu

The PMS is initialised using the commands which begin with upper case letters. Once all of the system processes have been initiated and put in place, only the eight commands starting with lower case letters are used. System initialisation involves starting the error server process and the daemons. A `hosts` file is maintained with the names of all participating host machines. The error server is started on any of these host machines with the daemons being started on all machines where user processes will be run. In Figure 10.4 an example configuration of the PMS using six host machines is shown.

Each of the system processes is shown as running on a separate processor. It is possible to place the command process and the error server on any of the host machines including those where the daemons are running. The only requirement is that a daemon process be placed on each host machine where user processes are to be run. After the system initialisation is performed, user

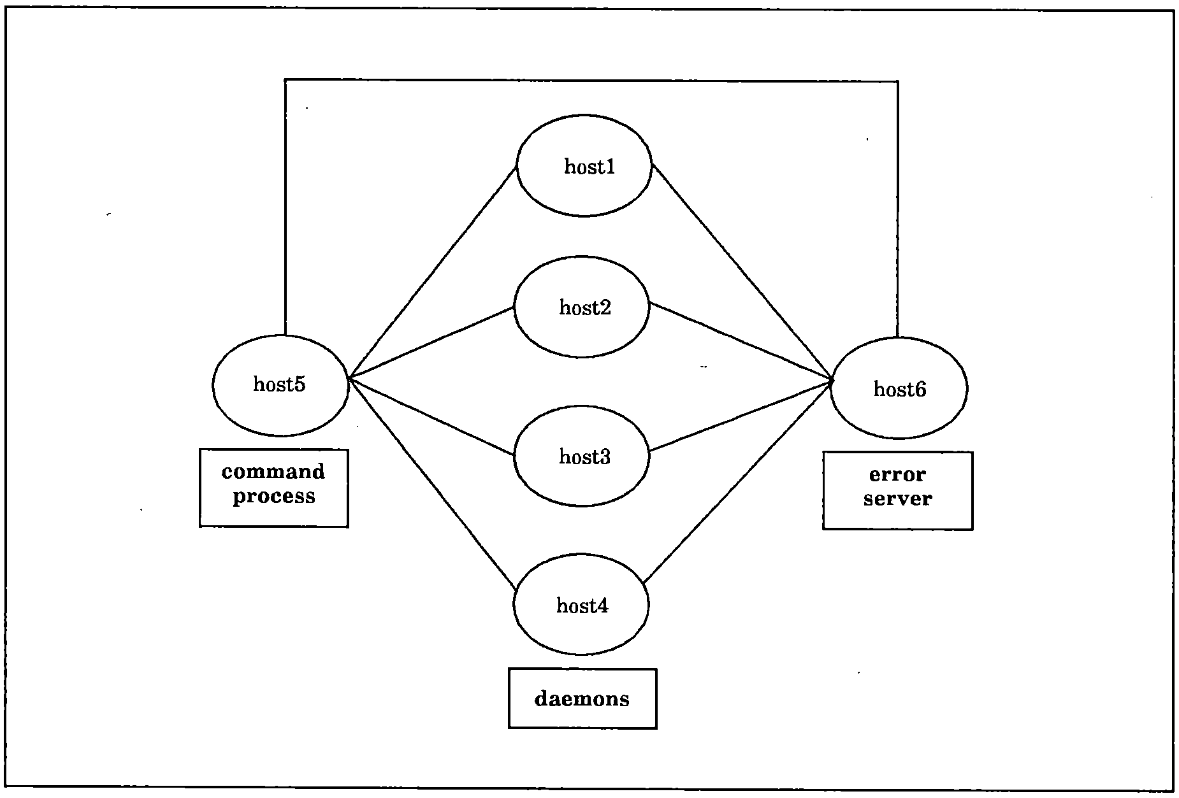


Figure 10.4 Example PMS Configuration

processes can be transported, started and examined on any of the host machines containing a daemon. These user processes are initialised and watched by daemons throughout their lifetime.

Each daemon maintains status information for any user processes executing on that host machine. This information is additionally relayed to the error server and the command process. Both the error server and the command process maintain a global view of all user processes on all host machines. Information is maintained on the time that each user process was started, and the current status of each process. The information is redundantly maintained so that if any single system process fails, the information on user processes is not lost.

10.3 Oracle Disassembler

The oracle disassembler is used to see exactly what intermediate instructions are generated from a source program. It is a menu driven disassembler entered by a Prolog command. The top-level menu for this tool can be seen in Figure 10.5.

The most helpful feature of this disassembler is option number four. This allows the user to specify a file where the Prolog source code for a predicate resides. This predicate is then located, compiled, loaded, and the low-level instructions displayed. The predicate can be edited externally,

```
                         options

      1   dump instructions to a file
      2   disassemble instructions starting at a specified location
      3   search for a symbol
      4   automatically compile, load and find a predicate
      5   print the index table for a predicate
      6   print instructions starting at the predicate entry point
      10  quit
```

Figure 10.5 Oracle Diassembler Top-Level Menu

and the user can again request option number four. Without ever leaving the disassembler, the user can see how changes made to the source code alter the low-level instructions.

10.4 Oracle Tracer

The oracle tracer is an application specific tool for following oracles throughout the Delphi machine. When this option is in use, the user is asked for the bit string describing the oracle to be traced. All ancestor and descendent oracles are then followed during the execution of the Prolog program. An ancestor string is any string which is a subset of the user given string. If the user requests a trace of the oracle [0110], then any communication involving a subset oracle [0], [01], [011], [0110] is reported. A descendent oracle is a bit string which has the user given oracle as a prefix. If the user requests a trace of the oracle [0110], then any oracle starting with the prefix [0110] (such as [0110101]) will be followed. The information provided by a trace is a listing of the ancestor or descendent oracle and the name of the host machine (and unique identifier for the Prolog process) where the oracle is being updated.

10.5 External Checkpointing

When a program is run over a large number of machines and for long periods of time (a few days to weeks), there are many outside complications which could result in the user losing solutions and having to restart the system from scratch. Reasons for having to restart the Delphi configuration range from machines crashing during the run to the logging files not being written because other users have consumed all of the available disk space. If the machines running the Prolog systems crash, Delphi automatically restarts the Prologs. Even this may not be desirable if the idea is to benchmark the Delphi system without including machine crashes. Also, the machine running the Controller may crash, and this is not handled internally by Delphi. For these reasons, a problem specific checkpointing facility has been developed for use in the Delphi project. This tool is the beginning of a more generalised checkpointing facility for UNIX processes.

Most of the fault tolerance built into the Controller is centred around checking on the status of Path Processors. If a Prolog system fails, the Controller is immediately notified by the receipt of an error message on that PP's socket connection. When this message is received, the Path Processor is down and is eliminated from the run. The oracle that the PP was last working on is held in the last run array, so it can be sent to an existing or a newly initialised PP. The host machine on which the PP was running is checked to see if it too has crashed. If not, a new PP is initialised on that same host. If the host machine crashing caused the error on the PP's socket connection, then a new host machine is found (if possible), and a new PP is started on the new host. If there are no more available host machines (in the configuration file), then either one of two things happens:

- An additional PP is started on a host machine already running a PP.
- The oracle in the last run array is held until one of the existing PPs reports in idle. It is then given to that idle PP.

It is easy to assure that crashing PPs do not cause any portion of the search space to be left unexplored. Even if all of the PPs crash at the same time, no information will be lost. There has never been a case where the Controller itself has crashed. The reason for this is that the Controller is usually run on a 'safe' machine; a machine which not many of the other users have access to. If the Controller did crash, all of the information about which oracles have been explored and which have not would be lost. This could be a catastrophic failure if Delphi were running a very large problem; many hours or days of work may have been lost. To avoid this problem, an external checkpointing facility has been developed in collaboration with a student at the Computer Laboratory. The design of this checkpointing facility is documented in Guest [1989].

The user of the checkpointing facility specifies how often the process is to be checkpointed. The state of that process is dumped at the specified intervals, and the process can be restarted from this state information. Currently, the checkpointing facility is not able to handle processes which have socket connections (this would involve also checkpointing the processes at the other end of the sockets). It is able to handle I/O to local files, and this is sufficient for checkpointing the Controller. Since the Controller maintains a last run array, no oracle will be lost. All of the PPs can be restarted, and the Delphi machine can continue from where it left off.

There are two ways to go about providing a checkpointing facility for Delphi. One method is to provide an external checkpointing facility usable on any UNIX process. The other solution is to have an internal, problem specific solution applicable only to the current Delphi machine implementation. An external method of checkpointing would involve having the process dump its state at certain intervals and then reestablish the communications necessary if a restart is to be done. At the present time, there is no system which can checkpoint an arbitrary UNIX process. It is a difficult problem, but would be a useful facility to have on an operating system.

A problem specific solution has already been accomplished by saving the contents of the last run array. The last run array maintains a copy of the last oracle sent to each of the Path Processors. From the configuration file and the contents of the last run array, the connections to host machines can be reestablished and the Path Processors restarted. This is a *coarse grain* solution since the Path Processors are restarted with the last oracles held by the Controller. It still may take a long time for the entire Delphi system to be back to the state it was in when he checkpoint was taken. For example, if the reassign jobs strategy is being used with a check-in interval of one thousand, then a great deal of work can easily be lost. A Path Processor (PP) may have been about to perform a check-in to the Controller just when the crash occurred. This loses the last one thousand steps of the computation performed by that PP since it was unable to communicate its updated position in the search space. The limit of this computation loss occurs with the automatic partitioning strategy. Checkpointing the system by utilising the last run array does not help at all with this strategy.

A *medium grain* solution is to have the PPs also keep track of their current position within the search space. When a checkpoint is taken, the PPs write out their current path, and a restart would start the processing from exactly this position within the search space. This solution is more difficult to implement since it involves a distributed checkpointing facility. Each of the checkpointed files would be on separate host machines, and a hardware failure would cause the loss of these files. This solution still involves the loss of some of the computation

The *fine grain* solution allows complete *process migration* with the loss of very little of the computation. Process migration is the term used to describe the ability of a process started on one processor to be migrated and started at some time in the future on another processor. This can be achieved by an external checkpointing facility which intercepts many of the operating system calls and saves the state of all processes.

A simple, problem specific approach was taken for the first checkpointing tool for the following reasons:

- The Controller already keeps track of the current oracle being explored on each of the Path Processors.
- No core files need to be kept for each of the Prologs. This reduces the complexity a great deal as there is no need to maintain redundant files for the Prolog systems. This avoids the need to maintain multiple copies of these files across the network in case of multiple crashes involving hardware failures. Only the Controller process has to make sure that its single file is safe.
- In the reassign jobs strategy, the Prologs are generally doing more work than the Controller, so the time taken by the Controller to perform the update is inconsequential. This might not be the case if the Prolog systems also had to be checkpointed.
- Only very simple code is needed to perform the timing (for the checkpoint interval) and dumping procedures; it only has to be applied to one process.

The only weakness to this solution is the fact that it might not be applicable to all future Delphi control strategies or implementations of the Delphi machine. For example, if multiple Controllers are maintained in a system with thousands or millions of Path Processors, the checkpointing system would need to be modified. With an external checkpointing facility (the fine grain solution), this would not be the case. To create a generalised UNIX checkpointing facility however, would require a great deal more work.

The current solution also needs more work to provide a safe checkpoint file. If the checkpoint file is held on the same machine which has crashed, then it is possible that the checkpoint file could be lost. If the machine has suffered a hardware failure then the checkpoint file may be unobtainable. To solve this problem, the checkpoint file must be redundantly sent to numerous host machines. Since the Delphi Controller never crashed throughout the benchmarking, these stringent measures never needed to be implemented.

10.6 Prolog Preprocessor

Nothing has been said about what Delphi does with extra-logical operators such as the cut operator, `assert` and `retract`. None of the benchmarked Prolog programs contained any of these extra-logical features; they purposely had been removed as can been seen by the `not_strict_member` predicate in the adder problem. If the benchmark programs had contained extra-logical features, they still could be run on the Delphi machine, but OR-parallelism would not be exploited in the predicates containing these features.

Delphi does not have any implementation of a parallel cut such as that described in Shapiro [1989]. If cuts are placed in any clauses within a predicate, then that predicate cannot be explored in parallel by Delphi. These predicates can, however, be sequentially explored by compiling and loading them for sequential execution.

The facilities for assuring sequential execution of a predicate already exist in the Delphi machine. The Prolog run-time system contains a great deal of code with extra-logical features in it, and this code is used by the Delphi machine. The Prolog system files have been compiled using the original SB-Prolog compiler, and they are loaded using the original SB-Prolog loader. Any user programs containing predicates which must be performed sequentially can be compiled and loaded with the original SB-Prolog system.

The Prolog preprocessor has been written to help in the task of locating and properly compiling predicates which must be performed sequentially. Two separate files are created if there are predicates which cannot be executed in parallel. One of these source Prolog files goes through the SB-Prolog system and the other is run through the Delphi compiler and loader. Both files are compiled by their respective compilers, and a single input file is created which has the proper specifications for loading each of the two compiled files. The predicates containing extra-logical features are not executed in parallel by Delphi, but they can be executed.

Chapter 11 Conclusions

Oracles have been shown to be a simple and effective way to control a parallel tree search among multiple uniprocessors communicating over a network. The use of oracles for exploiting OR-parallelism in Prolog programs has been demonstrated by various control strategies implemented on the Delphi machine. A number of Prolog programs were run on the Delphi machine and their execution times compared to other Prolog systems. The speed ups show that a distributed system using oracles to control the parallel search can be an efficient way to exploit the OR-parallelism in nondeterministic Prolog programs.

The speed ups obtained by Delphi are related to the amount and location of the OR-parallelism contained in the Prolog program. If there is no OR-parallelism to exploit, Delphi does not impose many overheads and the program runs at approximately the same speed as on the unmodified Prolog system (Cosmic Prolog). The worst case is a deterministic problem which contains *useless OR nodes*. Delphi is only indexed on the first argument of a clause, so a deterministic program containing a predicate with two or more clauses having the same first argument will create an OR node. This was seen in predicate mmc of the matrix multiplication problem:

```
mmc(_,[],[]).
mmc(A,[Bi|Bn],[Ci|Cn]) :-  ip(A,Bi,Ci), mmc(A,Bn,Cn).
```

The first argument in each clause will match anything so these two clauses will be in the same indexing set as defined by the Delphi indexing method. If indexing were performed on all three arguments in the head of these two clauses, each clause would be allocated to a separate indexing set. Most input clauses to this predicate will match only one of the two clause heads, but Delphi always creates an OR node and tries to match with both heads. Every time the predicate mmc is called, the Path Processor must perform all of the functions associated with reaching a choice point. These functions include:

- Maintenance of the *current path* data structure.
- Performing the calculations necessary to determine if a check-in should be performed.
- If a check-in is performed, the Path Processor must send a message to the Controller and wait for a response.
- The creation of a limited choice point.

For every OR node reached, these overheads cause an increase in the execution time. If there is no OR-parallelism to exploit, the addition of more processors to the Delphi configuration cannot compensate for the time lost in computational overheads. This is the case with deterministic programs which have OR nodes such as the matrix multiplication problem. Even with these overheads, the worst case performance of the Delphi machine is a slow down of one and a half times over the same problem run on Cosmic Prolog.

For the benchmarked Prolog programs, more OR nodes are explored in the nondeterministic programs than in the deterministic programs. Since more OR nodes are reached in the nondeterministic programs, more overheads are incurred causing an even greater increase in the execution time as compared to deterministic programs. This leads to a worst case performance for nondeterministic programs which is a slow down of two times over the same problem run on Comic Prolog. This slow down however, only occurs when running a single processor Delphi configuration. As more processors are added, OR-parallelism is exploited and a net decrease in the execution time can be seen. This leads to relative speed ups which vary according to the amount (and location) of OR-parallelism which can be exploited by Delphi. With a twenty processor Delphi configuration, the 8-queens problem shows a speed up of seven times (over the single processor Delphi configuration) while the 10-queens problem exhibits an eighteen times speed up. As compared to the unmodified Prolog system, the 10-queens problem shows a speed up of nine times.

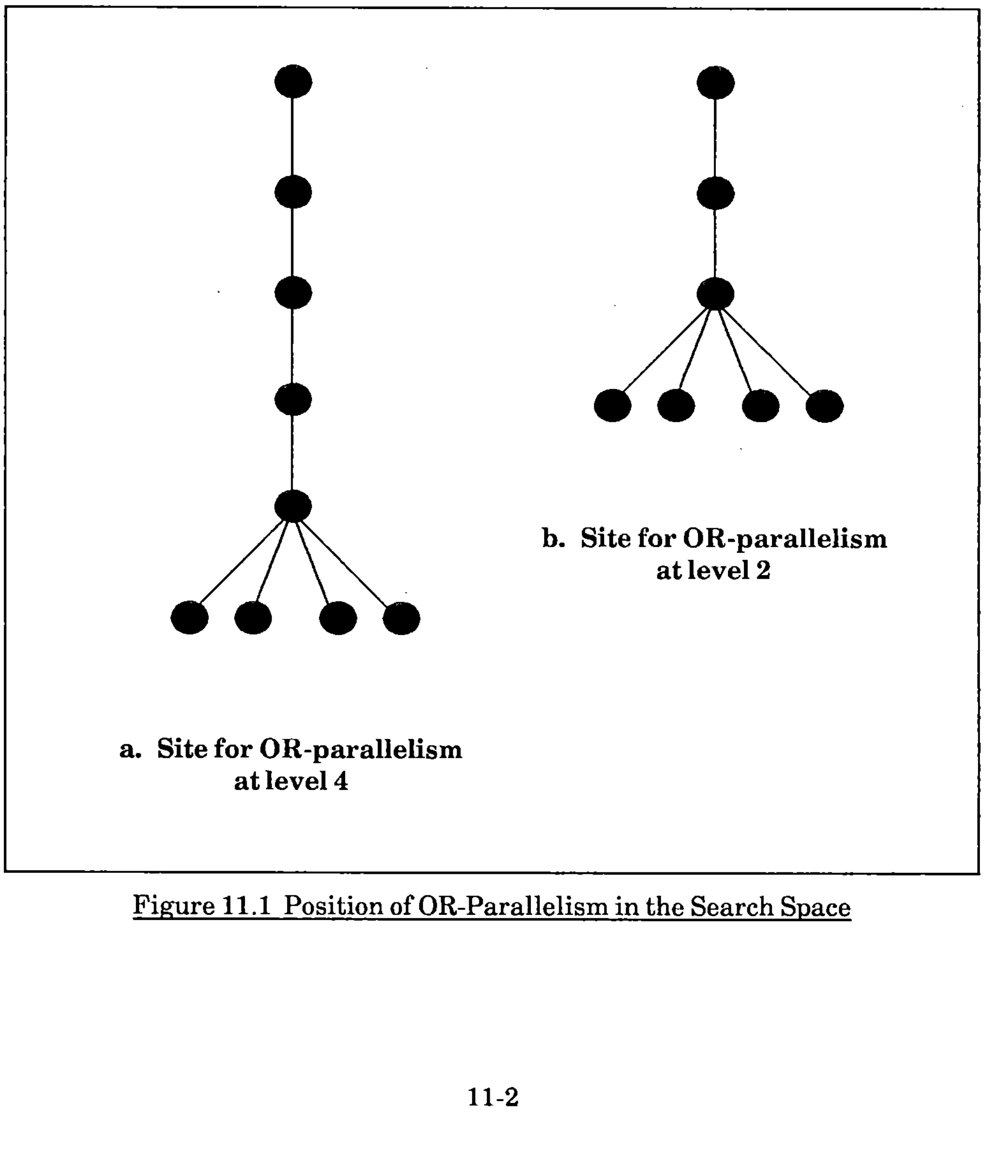

b. Site for OR-parallelism at level 2
a. Site for OR-parallelism at level 4

Figure 11.1 Position of OR-Parallelism in the Search Space

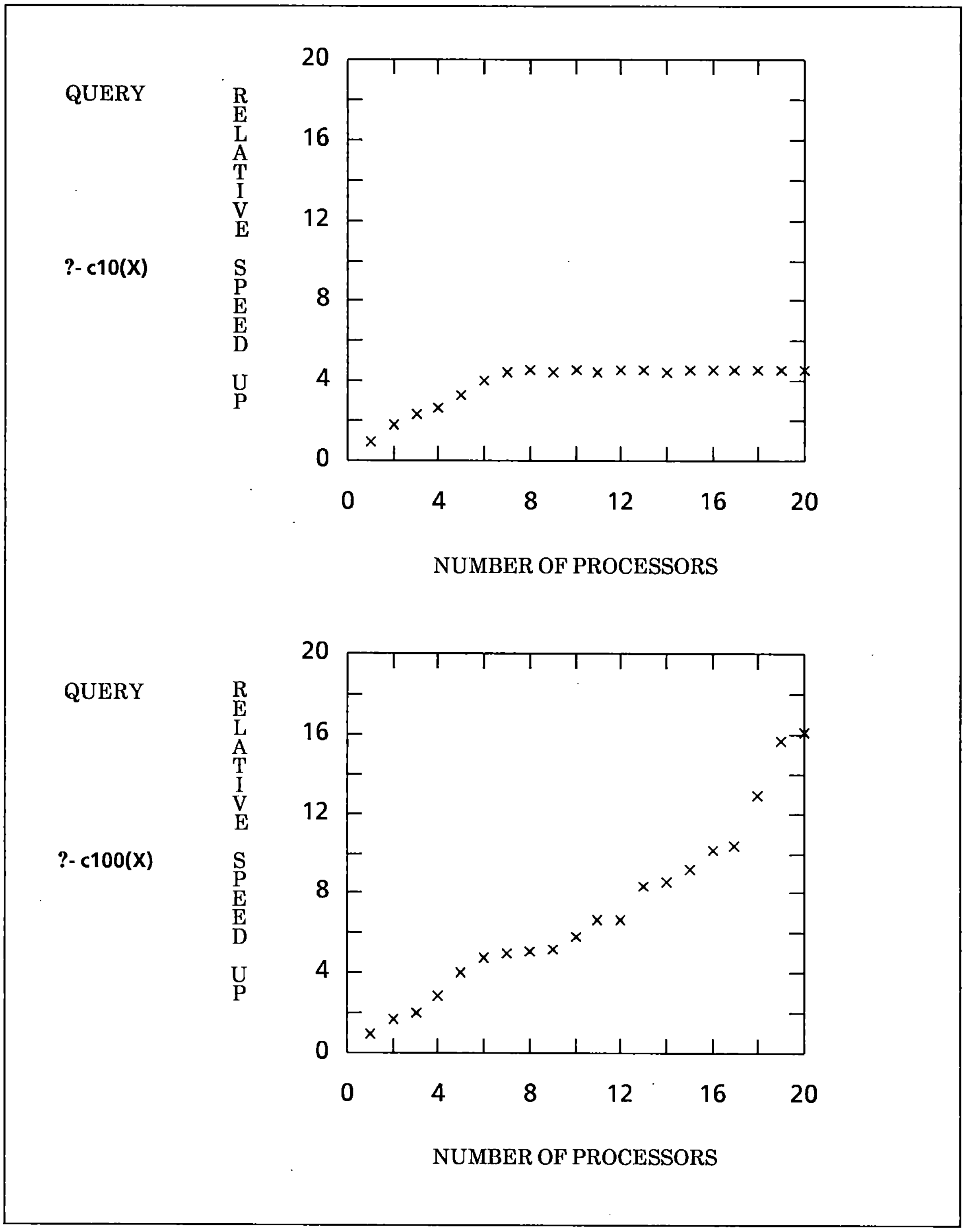


Figure 11.2 Relative Speed Up of Two Ortest Queries

The location of the OR nodes in the search space also affects the speed ups which Delphi can obtain. The higher-up the parallelism is in the tree, the less *redundant computation* occurs. Redundant computation occurs when any nodes in the search space are explored by more than one Path Processor. Delphi can exploit OR-parallelism more efficiently if it is higher up in the search space. This is demonstrated in Figure 11.1.

If we consider that each node in Figure 11.1 takes the same amount of time to explore, then a single processor Delphi configuration will explore the tree in Figure 11.1a in 9 time units. A Delphi configuration with two processors (using a central splitting algorithm) will explore the tree in 7 time units. The speed up is approximately 1.3. For the tree in Figure 11.1b, a single processor takes 7 and two processors 5 time units to explore the tree. The speed up in this case is 1.4. Though the amount of parallelism which can be exploited is the same, the location of the parallelism affects the speed ups which can be obtained by Delphi.

With the `ortest` queries `c10(X)` and `c100(X)`, the location of the OR-parallelism is held constant while the amount is increased by a factor of ten. These queries demonstrate how the quantity of OR-parallelism is related to the speed ups which can be obtained by Delphi (see Figure 11.2).

With the growth of personal workstations distributed on networks, many organisations have a large computing resource which often lies idle. Delphi provides a means of harnessing the spare power of this resource to speed up the execution of Prolog programs.

Appendix 4a Loaded Source

4a.1 Indexing Example Program

The following source Prolog program is a contrived example which demonstrates how clauses of a predicate are indexed and the use of oracle instructions. The intermediate code for this program is shown both for backtracking and non-backtracking strategies. Intermediate code contains the instructions generated when compiled code is loaded for execution on each of the Path Processors.

The source program contains the one predicate `oxx/2` consisting of 14 individual clauses with various types of first argument. Indexing will be performed on this first argument only. Notice that the clauses to be made into a set (a group having the same type of first argument) have not been grouped together. The order of the clauses as they appear in the source program does not affect the format of the indexing instructions which are generated. This specification leads to a very structured and easy to interpret intermediate code, which is not the case with all indexing systems. In some systems, the order of the clause's appearance in the predicate dictates what the layout of the instructions will be; a tidy format often relying on the assumption that programmers are careful about grouping together the clauses with a similar first argument.

```
oxx(X,1).
oxx(str(X,Y),1).
oxx(str(1,2),2).
oxx(V,a).
oxx([1,2],1).
oxx(V,b).
oxx([1,2,3],2).
oxx([],1).
oxx(V,c).
oxx([],2).
oxx(1,1).
oxx(5,[]).
oxx([],3).
oxx(5,a).
```

4a.2 Program Loaded for Non-backtracking

With non-backtracking strategies the only `onum` instruction needed is the `onumsing` instruction, as no choice points are ever created or destroyed. The following code shows the example program loaded and ready for execution using a non-backtracking control strategy. Bold type has been used for comments and are not a part of the intermediate instructions. These comments highlight the major divisions of the loaded program. A more detailed analysis of the indexing portion of this program is given in Section 4a.3.

Symbol Table

```
/* data below: name, arity, type, and entry point */

1fc58: _$interrupt/2,   ORDI,  0
1fc90: str/2,    ORDI,  0
1fc10: []/0,     ORDI,  0
1fc78: user/0,   ORDI,  0
1fcf0: oxx/2,    PRED,  1fdd2
1fca8: a/0,      ORDI,  0
1fcc0: b/0,      ORDI,  0
1fc40: ,/2,      ORDI,  0
1fcd8: c/0,      ORDI,  0
1fc28: ./2,      ORDI,  0
```

Clause Instructions for Procedure oxx/2

```
1fd00    getnumcon            2       1
1fd06    proceed
1fd08    getstr00             1       1fc90
1fd0e    unitvar00            1
1fd10    unitvar00            1
1fd12    getnumcon            2       1
1fd18    proceed
1fd1a    getstr00             1       1fc90
1fd20    uninumcon            1
1fd26    uninumcon            2
1fd2c    getnumcon            2       2
1fd32    proceed
1fd34    getcon00             2       1fca0
1fd3a    proceed
1fd3c    getlist00            1
1fd3e    uninumcon            1
1fd44    unitvar00            1
1fd46    getnumcon            2       1
1fd4c    getlist00            1
1fd4e    uninumcon            2
1fd54    uninil00
1fd56    proceed
1fd58    getcon00             2       1fcb8
1fd5e    proceed
1fd60    getlist00            1
1fd62    uninumcon            1
1fd68    unitvar00            1
1fd6a    getnumcon            2       2
1fd70    getlist00            1
1fd72    uninumcon            2
1fd78    unitvar00            2
1fd7a    getlist00            2
1fd7c    uninumcon            3
1fd82    uninil00
1fd84    proceed
1fd86    getnil00             1
1fd88    getnumcon            2       1
1fd8e    proceed
1fd90    getcon00             2       1fcd0
1fd96    proceed
1fd98    getnil00             1
1fd9a    getnumcon            2       2
1fda0    proceed
1fda2    getnumcon            1       1
1fda8    getnumcon            2       1
1fdae    proceed
1fdb0    getnumcon            1       5
1fdb6    getnil00             2
1fdb8    proceed
1fdba    getnil00             1
1fdbc    getnumcon            2       3
1fdc2    proceed
1fdc4    getnumcon            1       5
1fdca    getcon00             2       1fca0
1fdd0    proceed
```

```
Entry Point of Procedure oxx/2

1fdd2    jumponspecial          1fee8   1ff4c   1feae   1ffa2   2000e
Plenary set begins
1fde8    setmax                 e
1fdea    onumsing               2       1       1fcf0   1fd00
1fdf8    onumsing               2       2       1fcf0   1fd08
1fe06    onumsing               2       3       1fcf0   1fd1a
1fe14    onumsing               2       4       1fcf0   1fd34
1fe22    onumsing               2       5       1fcf0   1fd3c
1fe30    onumsing               2       6       1fcf0   1fd58
1fe3e    onumsing               2       7       1fcf0   1fd60
1fe4c    onumsing               2       8       1fcf0   1fd86
1fe5a    onumsing               2       9       1fcf0   1fd90
1fe68    onumsing               2       a       1fcf0   1fd98
1fe76    onumsing               2       b       1fcf0   1fda2
1fe84    onumsing               2       c       1fcf0   1fdb0
1fe92    onumsing               2       d       1fcf0   1fdba
1fea0    onumsing               2       e       1fcf0   1fdc4
Defaults set begins
1feae    setmax                 4
1feb0    onumsing               2       1       1fcf0   1fd00
1febe    onumsing               2       2       1fcf0   1fd34
1fecc    onumsing               2       3       1fcf0   1fd58
1feda    onumsing               2       4       1fcf0   1fd90
Nil set begins
1fee8    setmax                 7
1feea    onumsing               2       1       1fcf0   1fd00
1fef8    onumsing               2       2       1fcf0   1fd34
1ff06    onumsing               2       3       1fcf0   1fd58
1ff14    onumsing               2       4       1fcf0   1fd86
1ff22    onumsing               2       5       1fcf0   1fd90
1ff30    onumsing               2       6       1fcf0   1fd98
1ff3e    onumsing               2       7       1fcf0   1fdba
List set begins
1ff4c    setmax                 6
1ff4e    onumsing               2       1       1fcf0   1fd00
1ff5c    onumsing               2       2       1fcf0   1fd34
1ff6a    onumsing               2       3       1fcf0   1fd3c
1ff78    onumsing               2       4       1fcf0   1fd58
1ff86    onumsing               2       5       1fcf0   1fd60
1ff94    onumsing               2       6       1fcf0   1fd90
Structure set begins
1ffa2    switchonstructure      1ffac   3
1ffac           1feae
1ffb0           1ffb8
1ffb4           1feae
Subset of Structure set - clauses which unify with 'str' as first argument
1ffb8    setmax                 6
1ffba    onumsing               2       1       1fcf0   1fd00
1ffc8    onumsing               2       2       1fcf0   1fd08
1ffd6    onumsing               2       3       1fcf0   1fd1a
1ffe4    onumsing               2       4       1fcf0   1fd34
1fff2    onumsing               2       5       1fcf0   1fd58
20000    onumsing               2       6       1fcf0   1fd90
Integer set begins
2000e    switchoninteger        20018   5
20018           1feae
2001c           2002c
20020           20074
20024           1feae
20028           1feae
Subset of Integer set - clauses which unify with a '1' as first argument
2002c    setmax                 5
2002e    onumsing               2       1       1fcf0   1fd00
2003c    onumsing               2       2       1fcf0   1fd34
2004a    onumsing               2       3       1fcf0   1fd58
20058    onumsing               2       4       1fcf0   1fd90
20066    onumsing               2       5       1fcf0   1fda2
Subset of Integer set - clauses which unify with a '5' as first argument
20074    setmax                 6
20076    onumsing               2       1       1fcf0   1fd00
20084    onumsing               2       2       1fcf0   1fd34
20092    onumsing               2       3       1fcf0   1fd58
200a0    onumsing               2       4       1fcf0   1fd90
200ae    onumsing               2       5       1fcf0   1fdb0
200bc    onumsing               2       6       1fcf0   1fdc4
```

4a.3 Program Loaded for Backtracking

A portion of the same example program is shown loaded for use in a backtracking control strategy. This portion contains all of the instructions involved with the indexing of clauses. The three `onum` instructions `onumtry`, `onumretry`, and `onumtrust` are used instead of only `onumsing`, for sets containing more than one clause. In this program, all of the sets contain either no clauses (resulting in a pointer to the fail set or fail code) or multiple clauses. For this reason there are no `onumsing` instructions contained in the following code.

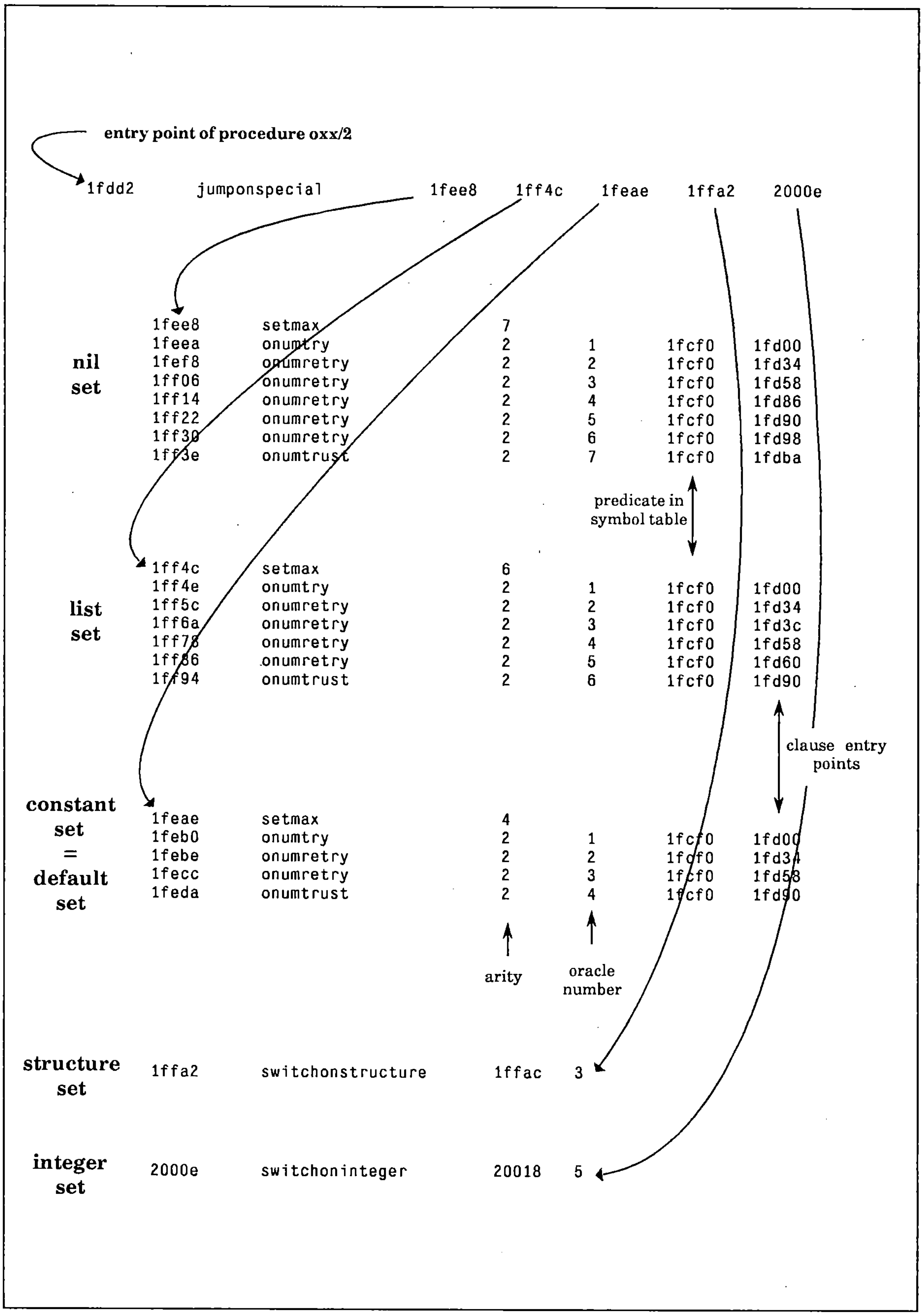
Figure 4a.1shows the `jumponspecial` instruction with pointers to the proper special sets. If a non-special clause needs to be unified, then the program counter falls through the `jumponspecial` instruction and the plenary set is used. The plenary set has not been shown in Figure 4a.1, but is similar to the plenary set code shown in Section 4a.2. Non-special or general clauses have a variable as their first argument. These general clauses will attempt to match all of the candidate clauses in the predicate (i.e. the plenary set).

The example program does not have any clauses containing a constant as their first argument. For this reason the pointer to the constant set (the third argument of `jumponspecial`) is actually a pointer to the default set. The default set contains all of the variable-keyed clauses in the predicate. These variable-keyed clauses will match any first argument. Notice that the entry points for clauses which are variable-keyed in the predicate are contained in all of the sets. If there had been no default set and no constant-keyed clauses in the predicate, then the third argument of `jumponspecial` would lead to the fail code.

Figure 4a.2 shows more detail on the `switchoninteger` instruction. This instruction begins the integer set for the example program. The `switchonstructure` and `switchonconstant` instructions are analogous to `switchoninteger` except that the address of the constant or structure (from the symbol table) is hashed instead of the integer.

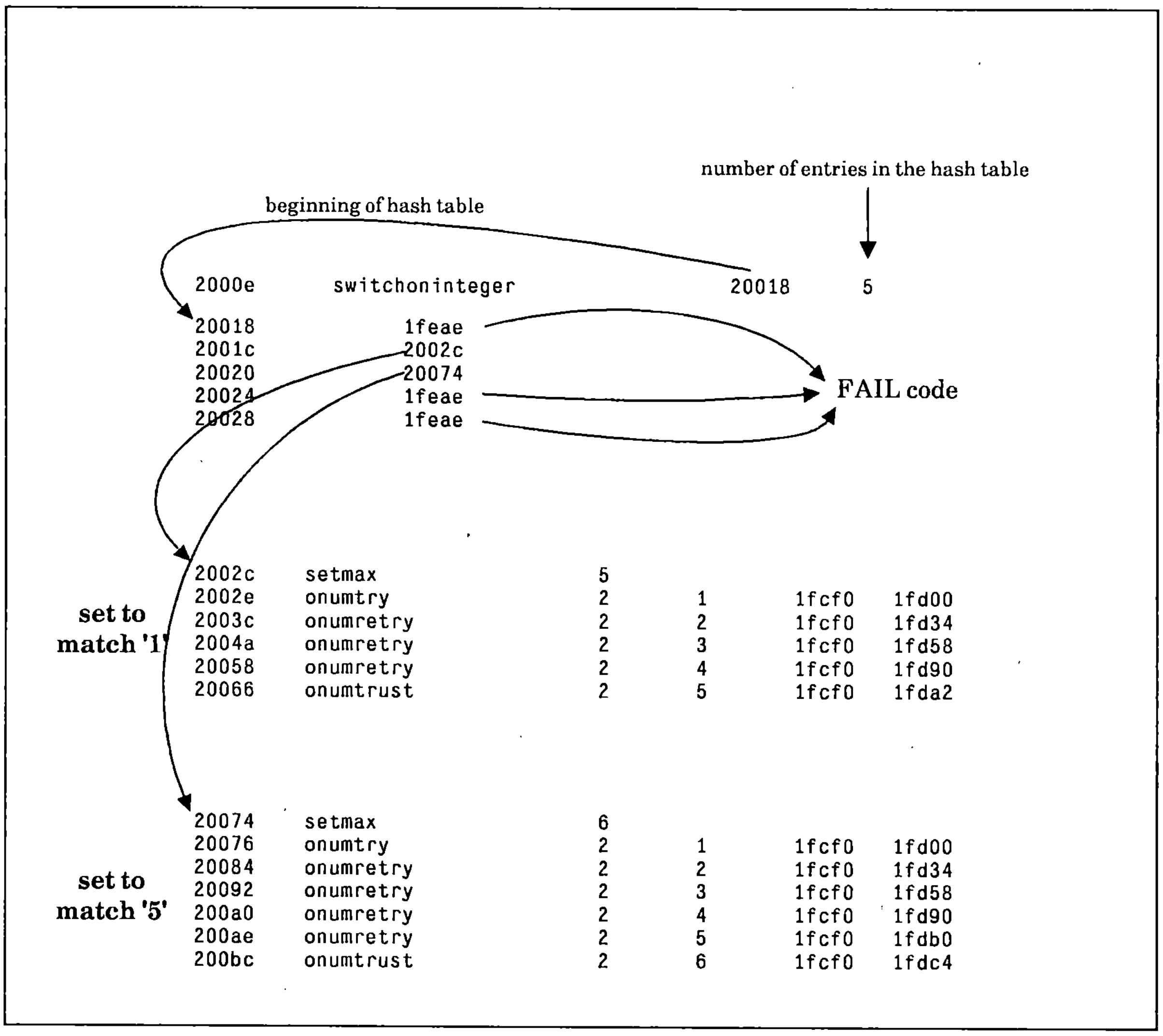
There are three clauses in the source program which have an integer as their first argument. Two of these clauses have the same integer, the number 5, as their first argument, and one clause the number 1. A hash table is created containing five buckets (buckets 0 through 4) into which these integer-keyed clauses will be hashed. Clauses grouped together in these buckets become the subsets pointed to by the entries in the hash table. For small integers, the value used in the hash function is the numeric value of the integer itself. The hash function used was:

$$((((\text{value}\ \&\ 0x3ffffffc) >> 2) + (\text{value}\ \&\ 0x3))\ \%\ \text{hash table size})$$


entry point of procedure oxx/2
nil set
list set
constant set = default set
structure set
integer set
predicate in symbol table
clause entry points
arity
oracle number

```
1fdd2     jumponspecial       1fee8   1ff4c   1feae   1ffa2   2000e

1fee8     setmax              7
1feea     onumtry             2       1       1fcf0   1fd00
1fef8     onumretry           2       2       1fcf0   1fd34
1ff06     onumretry           2       3       1fcf0   1fd58
1ff14     onumretry           2       4       1fcf0   1fd86
1ff22     onumretry           2       5       1fcf0   1fd90
1ff30     onumretry           2       6       1fcf0   1fd98
1ff3e     onumtrust           2       7       1fcf0   1fdba

1ff4c     setmax              6
1ff4e     onumtry             2       1       1fcf0   1fd00
1ff5c     onumretry           2       2       1fcf0   1fd34
1ff6a     onumretry           2       3       1fcf0   1fd3c
1ff78     onumretry           2       4       1fcf0   1fd58
1ff86     onumretry           2       5       1fcf0   1fd60
1ff94     onumtrust           2       6       1fcf0   1fd90

1feae     setmax              4
1feb0     onumtry             2       1       1fcf0   1fd00
1febe     onumretry           2       2       1fcf0   1fd34
1fecc     onumretry           2       3       1fcf0   1fd58
1feda     onumtrust           2       4       1fcf0   1fd90

1ffa2     switchonstructure   1ffac   3

2000e     switchoninteger     20018   5
```

Figure 4a.1 Instruction jumponspecial



```
2000e     switchoninteger              20018     5

20018          1feae
2001c          2002c
20020          20074
20024          1feae
20028          1feae

2002c    setmax           5
2002e    onumtry          2       1       1fcf0    1fd00
2003c    onumretry        2       2       1fcf0    1fd34
2004a    onumretry        2       3       1fcf0    1fd58
20058    onumretry        2       4       1fcf0    1fd90
20066    onumtrust        2       5       1fcf0    1fda2

20074    setmax           6
20076    onumtry          2       1       1fcf0    1fd00
20084    onumretry        2       2       1fcf0    1fd34
20092    onumretry        2       3       1fcf0    1fd58
200a0    onumretry        2       4       1fcf0    1fd90
200ae    onumretry        2       5       1fcf0    1fdb0
200bc    onumtrust        2       6       1fcf0    1fdc4
```

Figure 4a.2 Instruction switchoninteger

Using the hash function on the integer-keyed clauses of the example program sends the clauses with a first argument of '1' into bucket number 1. This bucket is the second entry in the hash table. Clauses with a first argument of '5' are put into bucket number 2. All other entries in the hash table lead to the fail code. When a pointer from the hash table is followed, it leads to a set containing the entry points of all clauses which have been hashed together, merged with all clauses contained in the default set. All clauses in any set are in the same order in which they appear in the source Prolog program.

As an example, assume that unification is attempted with the goal `oxx(5,X)`. First, the symbol table is searched for a predicate named `oxx` with arity 2. When this is found, the pointer corresponding to the entry point of `oxx/2` is followed leading to the `jumponspecial` instruction in Figure 4a.1. As the first argument of the goal is an integer, the fifth address of `jumponspecial` is placed in the program counter. The `switchoninteger` instruction of Figure 4a.2 is then encountered. The number 5 is hashed using the hash function. This new number is used as the offset from the first parameter of the `switchoninteger` instruction. Entry number three (the address 20074) is placed in the program counter, and the next instruction reached is a `setmax`. Oracle manipulations are performed at this point according to a particular control strategy.

Finally, for each clause to be executed, the offset from `setmax` is calculated, and an `onum` instruction is performed. Again this instruction is performed according to the current control strategy. The arity of the clause is given as the first parameter so that the proper number of arguments can be pushed and popped from the stack. The oracle number is provided so that a path from the root of the search space can be maintained. The third parameter shown here is for debugging purposes. It allows the name of the predicate (of which this clause is a part) to be derived within any `onum` instruction. The final parameter is the entry point to the clause instructions. The program counter is replaced by this address when a clause is to be tried.

5a.1 Amoeba Echo Server

This echo server written using Amoeba IPC demonstrates just how simple user-level communications across a network can be. Though this is a trivial example, it does include the transaction primitives used by any Amoeba server. When the server is initialised, it dynamically creates a unique port for it to listen on, (see ①) and places this unique port into the global directory space. Both the dynamic binding of a port name and advertising that name to any potential clients is accomplished by procedure `server_put_capability` (see Section 5a.3). Numbers ② and ③ show uses of the standard IPC commands used by the server to receive and send messages respectively.

```
#include "/usr/amoeba/h/amoeba.h"
#include <stdio.h>
#include <ctype.h>
#include <signal.h>
#include "shout.h"

extern capability *server_put_capability();

char Buf[BUFFER];
header CInBufHeader;


 main()
 {
     capability *p_svcap;
     char name[10];

     strcpy(name, "echo");
     p_svcap = server_put_capability(name);
     CInBufHeader.h_port = p_svcap->cap_port;      ①
     echo_server();
 }
```

① $\longrightarrow$ A capability (name/unique port pair) is created for the server. This new port name is copied into the Amoeba header.

```
 echo_server()
 {
     for (;;) {
         short   n, len;

         len = getreq (&CInBufHeader, Buf, BUFFER);      ②
         if (len < 0) {
             printf ("\necho_server: getreq failed len = %d\n", len);
             exit (0);
         }
         n = putrep (&CInBufHeader, Buf, len);           ③
         if (n < 0) {
             printf ("\necho_server: putrep failed, n = %d\n", n);
             exit (0);
         }
     }
 }
```

② $\longrightarrow$ `getreq` blocks on the port name in `CInBufHeader`

③ $\longrightarrow$ `putrep` sends `Buf` to the port name in `CInBufHeader`

5a.2 Amoeba Shout Client

The major transaction primitive in Amoeba is the trans call. This call will block until either the server with the port name specified in the header has been located, or the time specified in the timeout call has elapsed. If the server is located within the specified time, the trans continues to block until the server performs a putrep.

```
#include "/usr/amoeba/h/amoeba.h"
#include <ctype.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <stdio.h>
#include "shout.h"

extern capability *get_server_capability ();
#define         NILCAP          ((capability *) 0)

char Buf[BUFFER];
header COutBufHeader;

main()
{
    init_COutBufHeader();
    shout_it();
}


shout_it()
{
  int size, i, len, j;
  struct rusage before, after;

  for (size = 1; size <= END; size += STEP) {
    printf("size = %d, ", size);
    getrusage(RUSAGE_SELF, &before);
    for (i = 0; i < NUMTIMES; i++) {
      len = trans_C_output(size);
      if (len != size) {
          fprintf(stderr, "shout: Fragmentation (size = %d, len = %d)\n",size, len);
          exit(1);
      }
      for (j = 0; j < DELAY; j++)
        ;
    }
    getrusage(RUSAGE_SELF, &after);
    Summary(&before, &after);
  }
}


Summary(before, after)
  struct rusage *before, *after;
{
  long sec;
  long usec;

  sec = after->ru_utime.tv_sec - before->ru_utime.tv_sec;
  usec = after->ru_utime.tv_usec - before->ru_utime.tv_usec;
  if (usec < 0) { sec--; usec += 1000000; }
  printf("User-time = %ld.%06ld, ", sec, usec);

  sec = after->ru_stime.tv_sec - before->ru_stime.tv_sec;
  usec = after->ru_stime.tv_usec - before->ru_stime.tv_usec;
  if (usec < 0) { sec--; usec += 1000000; }
  printf("System-time = %ld.%06ld\n", sec, usec);
}
```

```
init_COutBufHeader()
{
    capability *p_cap;

    while ((p_cap = get_server_capability("echo")) == NILCAP) {
        printf("\ninit_COutBufHeader: please start the echo server!\n");
        sleep(10);
    }
    COutBufHeader.h_port = p_cap -> cap_port;
}


int trans_C_output(size)
  int size;
{
    short   n;

    timeout (100);
    do {
        n = trans (&COutBufHeader, Buf, size, &COutBufHeader, Buf, BUFFER);
        if (n < 0) {
            printf ("\ntrans_C_output: trans failed\n");
            switch (-n) {
                case 1:
                    printf ("FAIL because of network or server crash\n");
                    exit(0);
                    break;
                case 2:
                    printf ("trans_C_output: NOTFOUND trans timed out\n");
                    break;
                case 3:
                    printf ("trans_C_output: BADADDRESS address of buffer is invalid\n");
                    exit (0);
                    break;
                case 4:
                    printf ("trans_C_output: ABORTED kernal aborted trans interrupt\n");
                    exit (0);
                    break;
                default:
                    printf ("trans_C_output: unknown trans error n = %d\n", n);
                    exit (0);
                    break;
            }                   /* end of switch */
        }                       /* end if n < 0 */
    } while (n < 0);
    return(n);
}
```

5a.3 Capability Functions

The following code allows servers to dynamically create unique port names and to advertise these ports in the global directory space (see procedure server_put_capability). The client can find a dynamically created port by searching for the object name in the global directory space (see procedure get_server_capability).

```
#include "/usr/amoeba/h/amoeba.h"
#include <stdio.h>
#include <ctype.h>
#include <signal.h>

#define         NILCAP          ((capability *) 0)

capability *server_put_capability(path)
  char path[];
{
    char host[20];
    capability *p_cap;
    capability dummy;
    register n;
    int count;

    if ( (p_cap = (capability *) malloc(sizeof(capability))) == NULL){
        printf("\nserver_put_capability: malloc failed\n");
        return(NULL);
    }
    gethostname(host,20);
    uniqport(&p_cap->cap_port);
    uniqport(&p_cap->cap_priv.prv_random);
    printf("\nin server with path %s\n\n", path);
    if ((n = am_lookup(path, &dummy)) == 0){ /* previous entry exists */
        if ((n = am_delete(path)) != 0){
            printf("\ndirectory %s was found but could not be deleted\n", host);
            exit(0);
        }
    }
    count = 0;
    do {
        count++;
        if ((n = am_append(path, p_cap)) != 0){
            printf("\nrep: am_append failed appending path %s, error was %d\n", path, n);
        }
        if (count > 100) printf("server_put_capability: tried to am_append 100 times!");
    } while(n < 0);
    return(p_cap);
} /* end server_put_capability */


capability *get_server_capability(path)
  char path[];
{
    short   n;
    capability *p_cap;

    if ( (p_cap = (capability *) malloc(sizeof(capability))) == NULL){
        printf("\nget_server_capability: malloc failed\n");
        return(NULL);
    }
    if ((n = am_lookup (path, p_cap) != 0)) {
        free(p_cap);
        return (NILCAP);
    }
    else {
        return (p_cap);
    }
} /* end get_server_capability */
```

5a.4 Sockets Echo Server

This echo server uses 4.2BSD socket connections for communication.

```
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <netdb.h>
#include <errno.h>
#include "shout.h"


void Copy(src, dst)
  int src, dst;
{
  char buffer[BUFFER];
  int n;

  while ((n = read(src, buffer, BUFFER)) > 0) {
    /* The assumption is made that if no error */
    /* occurs during the following write then  */
    /* all characters have been written; this  */
    /* may not necessarily be true.            */
    if (write(dst, buffer, n) < 0) {
      perror("echos: write()");
      exit(1);
    }
  }
  if (n < 0) {
    perror("echos: read()");
    exit(1);
  }
}


void Mourn()
{
  union wait status;

  while (wait3(&status, WNOHANG, 0) >= 0)
    ;
}


struct sockaddr_in myaddr_in;    /* Defaults to all zero */
struct sockaddr_in peeraddr_in;  /* Defaults to all zero */

int main(argc, argv)
  int argc;
  char *argv[];
{
  struct servent *sp;
  int ls;

  if (argc != 2) {
    fprintf(stderr, "Usage:  echos <service>\n");
    exit(1);
  }

  sp = getservbyname(argv[1], "tcp");
  if (sp == 0) {
    fprintf(stderr, "echos: Service `%s' not in /etc/services\n", argv[1]);
    exit(1);
  }

  myaddr_in.sin_family = AF_INET;
  myaddr_in.sin_addr.s_addr = INADDR_ANY;
  myaddr_in.sin_port = sp->s_port;
```

```
    ls = socket(AF_INET, SOCK_STREAM);
    if (ls < 0) {
      perror("echos: socket()");
      exit(1);
    }

    if (bind(ls, &myaddr_in, sizeof(myaddr_in)) < 0) {
      perror("echos: bind()");
      exit(1);
    }

    if (listen(ls, 5) < 0) {
      perror("echos: listen()");
      exit(1);
    }

    switch (fork()) {
      case -1:
        perror("echos: fork()");
        exit(1);
      case 0:
        signal(SIGCHLD, Mourn);
        for (;;) {
          int addrlen = sizeof(peeraddr_in);
          int s = accept(ls, &peeraddr_in, &addrlen);
          if (s < 0) {
            if (errno != EINTR) {
              perror("echos: accept()");
              exit(1);
            }
          } else {
            switch (fork()) {
              case -1:
                perror("echos: fork()");
                exit(1);
              case 0:
                close(ls);
                Copy(s, s);
                exit(0);
              default:
                close(s);
            }
          }
        }
      default:
        exit(0);
    }

    return 0;
  }
```

5a.5 Sockets Shout Client

This client uses the 4.2BSD socket connections for communication to the echo server. Once a connection to the server has been established, the client enters a loop to transmit various sized packets to the echo server.

```
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <netdb.h>
#include "shout.h"


void Mourn()
{
  union wait status;

  while (wait3(&status, WNOHANG, 0) >= 0)
    ;
}


void Summary(before, after)
  struct rusage *before, *after;
{
  long sec;
  long usec;

  sec = after->ru_utime.tv_sec - before->ru_utime.tv_sec;
  usec = after->ru_utime.tv_usec - before->ru_utime.tv_usec;
  if (usec < 0) { sec--; usec += 1000000; }
  printf("User-time = %ld.%06ld, ", sec, usec);

  sec = after->ru_stime.tv_sec - before->ru_stime.tv_sec;
  usec = after->ru_stime.tv_usec - before->ru_stime.tv_usec;
  if (usec < 0) { sec--; usec += 1000000; }
  printf("System-time = %ld.%06ld\n", sec, usec);
}


struct sockaddr_in myaddr_in;     /* Defaults to all zero */
struct sockaddr_in peeraddr_in;   /* Defaults to all zero */

int main(argc, argv)
  int argc;
  char *argv[];
{
  struct hostent *hp;
  struct servent *sp;
  int s, size, n, i, j;
  char buffer[BUFFER];
  struct rusage before, after;

  if (argc != 3) {
    fprintf(stderr, "Usage:  shout <host> <service>\n");
    exit(1);
  }

  hp = gethostbyname(argv[1]);
  if (hp == 0) {
    fprintf(stderr, "shout: Host `%s' not in /etc/hosts\n", argv[1]);
    exit(1);
  }
  sp = getservbyname(argv[2], "tcp");
  if (sp == 0) {
    fprintf(stderr, "shout: Service `%s' not in /etc/services\n", argv[2]);
    exit(1);
  }

  peeraddr_in.sin_family = AF_INET;
  peeraddr_in.sin_addr.s_addr = ((struct in_addr *)(hp->h_addr))->s_addr;
  peeraddr_in.sin_port = sp->s_port;

  s = socket(AF_INET, SOCK_STREAM);
  if (s < 0) {
    perror("shout: socket()");
    exit(1);
  }

  if (connect(s, &peeraddr_in, sizeof(peeraddr_in)) < 0) {
    perror("shout: connect()");
    exit(1);
  }

  signal(SIGCHLD, Mourn);

  for (size = 1; size <= END; size += STEP) {
    printf("size = %d, ", size);
    getrusage(RUSAGE_SELF, &before);
    for (i = 0; i < NUMTIMES; i++) {
      if (write(s, buffer, size) < 0) {
        perror("shout: write()");
        exit(1);
      }
      n = read(s, buffer, BUFFER);
      if (n < 0) {
        perror("shout: read()");
        exit(1);
      } else if (n != size) {
        fprintf(stderr, "shout: Fragmentation (size = %d, n = %d)\n", size, n);
        exit(1);
      }
      for (j = 0; j < DELAY; j++)
        ;
    }
    getrusage(RUSAGE_SELF, &after);
    Summary(&before, &after);
  }
  shutdown(s, 2);  /* No more sends or recvs */

  return 0;
}
```

5a.6 Mutual Header File

This is the shout.h header file used by both versions (Amoeba and Sockets) of the echo server and shout client.

```
#define         BUFFER          100
#define         STEP            1
#define         NUMTIMES        1000
#define         START           0
#define         END             100
#define         DELAY           0
```

Appendix 5b A Better Way to Communicate?

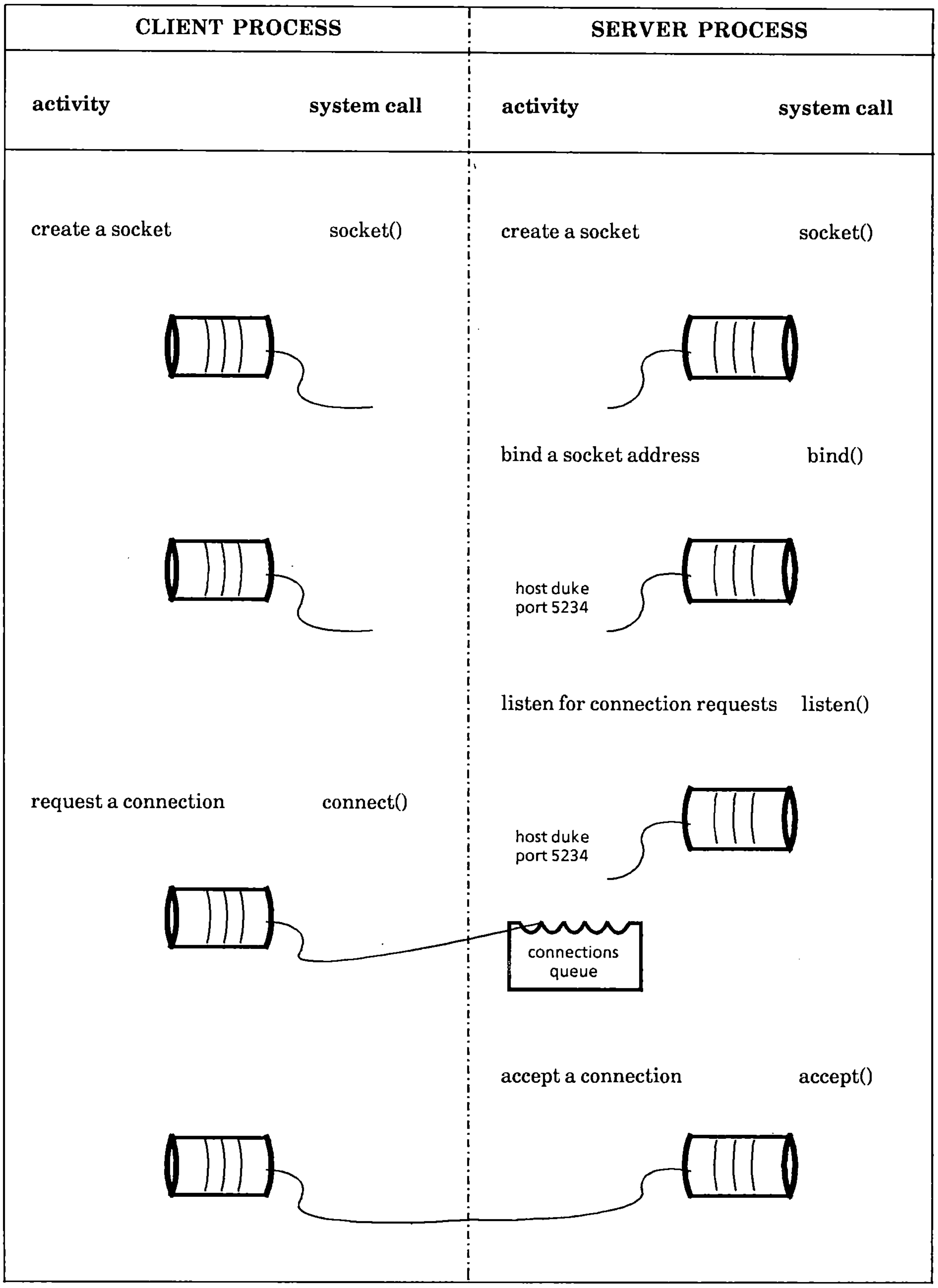


Figure 5b.1 Making a Connection

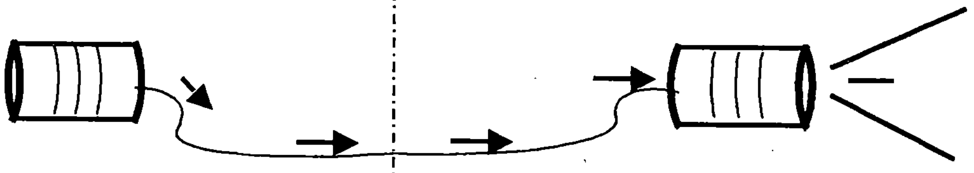
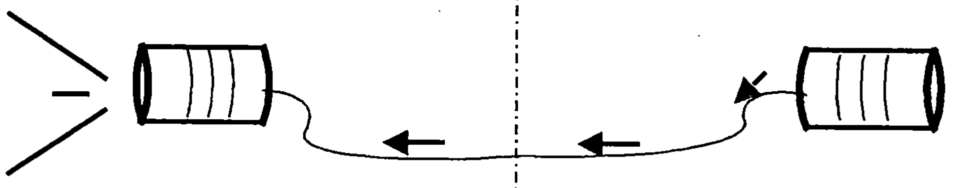
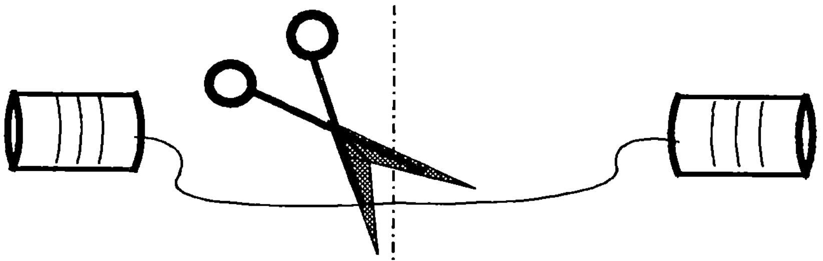
CLIENT PROCESS		SERVER PROCESS	
activity	**system call**	**activity**	**system call**
initiate communications	write()	receive a message	read()
			
receive server's response	read()	send a reply	write()
			
end a connection	close() or shutdown()	end a connection	close() or shutdown()
			

Figure 5b.2 Using a Connection

Appendix 6a Global Log File

All answers and other messages produced during the execution of a Prolog program are collected in a global log file. This file is located on the same host machine on which the Controller is started. The Delphi configuration file does not list the name of the host machine on which the Controller executes. The Controller can be started on any host machine and one possibility is to run it with one or more Prologs executing on the same processor. The only way to tell precisely how the configuration was set up is to look at the global log file produced.

6a.1 Example Configuration

Section 6a.2 is a portion of the global log file from a run of the 10-queens problem on ten host processors. The configuration file used for the example run was the following:

```
cheetah
eyra
jaguar
leopard
lynx
margay
ocelot
panther
puma
serval
```

As no number of Prologs has been shown explicitly for any of the host machines, the default number of one Prolog process per machine is used. In the example run the Controller process executed on the first host machine, named `cheetah`, and all of the host machines were used (each ran a Prolog process). In the start up command to the Controller it is possible to specify a particular number of host machines to be used for a run which is smaller than the number of entries in the configuration file. Information about which host machines are used in the execution of a program is contained in the global log file.

6a.2 Example Global Log File

```
Wed Jun 21 09:46:15 1989  puma  prolog_same: accepted client connection on fd 3 Socket 22

Wed Jun 21 09:46:15 1989  panther  prolog_same: accepted client connection on fd 3 Socket 20

Wed Jun 21 09:46:14 1989  ocelot  prolog_same: accepted client connection on fd 3 Socket 18

Wed Jun 21 09:46:13 1989  margay  prolog_same: accepted client connection on fd 3 Socket 16

Wed Jun 21 09:46:12 1989  lynx  prolog_same: accepted client connection on fd 3 Socket 14

Wed Jun 21 09:46:11 1989  leopard  prolog_same: accepted client connection on fd 3 Socket 12

Wed Jun 21 09:46:09 1989  jaguar  prolog_same: accepted client connection on fd 3 Socket 10

Wed Jun 21 09:46:09 1989  eyra  prolog_same: accepted client connection on fd 3 Socket 8

Wed Jun 21 09:46:08 1989  cheetah  prolog_same: accepted client connection on fd 3 Socket 6

Wed Jun 21 09:46:19 1989  serval  prolog_same: accepted client connection on fd 3 Socket 24

Wed Jun 21 09:46:22 1989  Controller on host cheetah, answer_sock: START TIMING

Wed Jun 21 09:46:25 1989  eyra  ANSWER  X =
[square(10,6),square(9,9),square(8,7),square(7,1),square(6,4),square(5,2),square(4,5),
square(3,8),square(2,10),square(1,3)] Prolog socket 8

Wed Jun 21 09:46:25 1989  cheetah  ANSWER  X =
[square(10,9),square(9,6),square(8,2),square(7,7),square(6,1),square(5,3),square(4,5)
,square(3,8),square(2,10),square(1,4)] Prolog socket 6

Wed Jun 21 09:46:25 1989  cheetah  ANSWER  X =
[square(10,7),square(9,9),square(8,2),square(7,6),square(6,1),square(5,3),square(4,5)
,square(3,8),square(2,10),square(1,4)] Prolog socket 6

Wed Jun 21 09:46:25 1989  leopard  ANSWER  X =
[square(10,3),square(9,5),square(8,8),square(7,1),square(6,9),square(5,4),square(4,2)
,square(3,7),square(2,10),square(1,6)] Prolog socket 12

Wed Jun 21 09:46:26 1989  jaguar  ANSWER  X =
[square(10,6),square(9,4),square(8,2),square(7,7),square(6,9),square(5,3),square(4,1),
square(3,8),square(2,10),square(1,5)] Prolog socket 10

Wed Jun 21 09:46:26 1989  serval  ANSWER  X =
[square(10,7),square(9,4),square(8,6),square(7,1),square(6,9),square(5,5),square(4,3),
square(3,8),square(2,10),square(1,2)] Prolog socket 24

Wed Jun 21 09:46:28 1989  ocelot  ANSWER  X =
[square(10,2),square(9,4),square(8,6),square(7,8),square(6,10),square(5,1),square(4,3)
,square(3,5),square(2,7),square(1,9)] Prolog socket 18

Wed Jun 21 09:46:28 1989  panther  ANSWER  X =
[square(10,4),square(9,7),square(8,9),square(7,2),square(6,6),square(5,1),square(4,3)
,square(3,5),square(2,8),square(1,10)] Prolog socket 20

Wed Jun 21 09:46:29 1989  jaguar  ANSWER  X =
[square(10,1),square(9,7),square(8,9),square(7,3),square(6,8),square(5,2),square(4,4),
square(3,6),square(2,10),square(1,5)] Prolog socket 10

Wed Jun 21 09:46:29 1989  leopard  ANSWER  X =
[square(10,9),square(9,4),square(8,2),square(7,8),square(6,3),square(5,1),square(4,7)
,square(3,5),square(2,10),square(1,6)] Prolog socket 12

Wed Jun 21 09:46:29 1989  panther  ANSWER  X =
[square(10,3),square(9,6),square(8,9),square(7,7),square(6,1),square(5,4),square(4,2)
,square(3,5),square(2,8),square(1,10)] Prolog socket 20

Wed Jun 21 09:46:29 1989  jaguar  ANSWER  X =
[square(10,8),square(9,1),square(8,3),square(7,9),square(6,7),square(5,2),square(4,4),
square(3,6),square(2,10),square(1,5)] Prolog socket 10

Wed Jun 21 09:46:29 1989  lynx  ANSWER  X =
[square(10,5),square(9,9),square(8,2),square(7,4),square(6,8),square(5,1),square(4,3),
square(3,6),square(2,10),square(1,7)] Prolog socket 14

Wed Jun 21 09:46:29 1989  panther  ANSWER  X =
[square(10,6),square(9,3),square(8,9),square(7,7),square(6,1),square(5,4),square(4,2)
,square(3,5),square(2,8),square(1,10)] Prolog socket 20

Wed Jun 21 09:46:30 1989  margay  ANSWER  X =
[square(10,4),square(9,9),square(8,7),square(7,3),square(6,1),square(5,6),square(4,2),
square(3,5),square(2,10),square(1,8)] Prolog socket 16

                                        •

                                        •

                                        •

Wed Jun 21 09:52:06 1989  serval  ANSWER  X =
[square(10,5),square(9,8),square(8,2),square(7,4),square(6,10),square(5,7),square(4,9)
,square(3,6),square(2,3),square(1,1)] Prolog socket 24

Wed Jun 21 09:52:06 1989  serval  ANSWER  X =
[square(10,8),square(9,5),square(8,2),square(7,4),square(6,10),square(5,7),square(4,9)
,square(3,6),square(2,3),square(1,1)] Prolog socket 24
```

```
Wed Jun 21 09:52:08 1989  serval  ANSWER  X =
[square(10,7),square(9,4),square(8,2),square(7,9),square(6,5),square(5,10),square(4,8)
,square(3,6),square(2,3),square(1,1)] Prolog socket 24

Wed Jun 21 09:52:18 1989  eyra  ANSWER  X =
[square(10,7),square(9,2),square(8,4),square(7,8),square(6,10),square(5,5),square(4,9),
square(3,6),square(2,1),square(1,3)] Prolog socket 8

Wed Jun 21 09:52:32 1989  Controller  PROBLEM HAS BEEN SOLVED time 369.440

Wed Jun 21 09:52:32 1989  Controller  Total Number of Checkins = 0
```

6a.3 Explanation of the Log File

Each entry in the global log file is time-stamped on the machine which originated the message. The message is then logged into a local file and also sent over the network to the global log file. These messages do not necessarily arrive in chronological order as seen in the first ten lines of the global log file. The first socket connection was made from the Controller to the host machine cheetah as the time stamp shows (the ninth message in the global log file). The Controller makes connections to the host machines in order of the configuration file entries.

After the initial connections are made to the ten participating host machines, the execution time clock is started. What is being timed is the sending of initialisation parameters to each of the host machines, the start-up of the Prolog system (including the reading of all the system start-up files), answers being time-stamped and sent over the network, and all control communications. In the given example, the strategy being employed is *automatic partitioning only* so there are no control communications. The fact that no control communications took place within this run is shown as the last entry in the file.

By far the major portion of a global log file is taken up by the answers sent from the Prolog processes. The actual log file (of which this is an excerpt) contains 724 "answer" messages as there are 724 ways to place 10 queens on a 10 X 10 chess board so that no queen is being attacked. The vertical ellipsis has been used in the example log file showing the location of the numerous omitted answer messages. The complete log file has not been shown because of space considerations. Table 6a.1 shows the number of answers found for each processor running the 10-queens problem. The number of answers found per processor for the 8-queens problem is shown for comparison.

It is important to realise that the global log file is a convenience for the user of Delphi. It allows the user to see all of the solutions to the problem as soon as they become available. If it were imperative to keep down the communications traffic across the network, this global log file need not be created at all. Since the local log files on each of the host machines maintain a duplicate copy of all answers, it is not mandatory for the global file to additionally be updated. The user could at "off peak" times (when network traffic and load on the host machines is low) collect the answers from each of the local log files and recreate the global log file from each of these pieces. In fact, with the automatic partitioning strategy used on its own, it is possible to run each of the host

host machine name	number of answers found 8-queens problem	number of answers found 10-queens problem
cheetah	8	93
eyra	0	65
jaguar	4	92
leopard	18	92
lynx	16	93
margay	0	65
ocelot	18	48
panther	4	64
puma	16	0
serval	8	112

Table 6a.1 Distribution of Answers

machines independently. There is no need for them to be working simultaneously on the problem at all. The two initial parameters needed to run automatic partitioning could be hard-wired or read from a local file. What this means is that the processors could be used whenever and only when their load falls below some particular level. Running Delphi is this manner means that it need never interfere with the functioning of other users on the network or on any of the host machines. The Prolog processes could go about finding their solutions without causing any network traffic congestion and only using the "spare" CPU cycles available on the host machines.

Appendix 6b Sessions

Three Delphi sessions are shown in this appendix. These sessions demonstrate the three methods of using the Controller to establish connections to Prolog processes across the network. In all three sessions it is assumed that the necessary files have been updated and placed in their proper directories on each of the host machines. The default input file, `prolog_standard_input`, was used in all three of these sessions. This file contained the following:

```
$load('buf_io').
$load('oracle').
orc_load('queens.del').
$load('mytop').
mytop.
get_solutions(4,X).
```

The configuration desired in every case was to use five host machines with eight Prolog processes running on them. Host `path01` with three Prologs, `path02`, `duke` and `fylde` each with one Prolog, and host `hendy` with two Prolog processes.

6b.1 Direct Connect to Prologs

Initiating Prolog processes by the direct connect method is labour intensive. The user must log onto each of the host machines and start every Prolog manually. Each Prolog must be interactively assigned a port (service) name then set into the background. When the user logs off the host machine the Prologs are not disturbed since they capture any signals which might otherwise terminate them. After the Prolog processes have been started, they listen on a port waiting for a client process to make a connection. The Controller is used with a menu driven program to establish these direct Prolog connections.

When the Controller is invoked with the command `Controller` and no command line arguments, an interactive program is entered. The direct connect facility is used to establish connections to all of the Prologs which are used in the configuration. The global log file is listed after the Controller is exited.

```
delphi: init: rlogin path01
Last login: Sat Sep 16 16:20:44 from delphi
Ultrix V2.2 System #285: Tue Sep  5 16:52:32 BST 1989


                Digital Equipment Corporation
                Merrimack, New Hampshire.

path01: csk: cd DelNet
path01: DelNet: prolog

init_connection_info: service for direct Prolog connection? test3
<<< direct Prolog Connection >>>
^Z
Stopped
path01: DelNet: bg
[1]  prolog &
path01: DelNet: prolog
```

```
init_connection_info: service for direct Prolog connection? test0
<<< direct Prolog Connection >>>
^Z
Stopped
path01: DelNet: bg
[2]  prolog &
path01: DelNet: prolog

init_connection_info: service for direct Prolog connection? test1
<<< direct Prolog Connection >>>
^Z
Stopped
path01: DelNet: bg
[3]  prolog &
path01: DelNet:
/usr/users/csk/.finish: not found
Connection closed

delphi: init: rlogin path02
Last login: Sat Sep 16 16:22:44 from delphi
Ultrix-32 V3.0 (Rev 63) System #7: Wed Aug 16 17:28:04 BST 1989
Welcome to path02.

**** This system is running testing auto NFS mount code (16). pb 24 Aug 89 ****


Terminal type = xterm
path02: csk: cd DelNet
path02: DelNet: prolog

init_connection_info: service for direct Prolog connection? test0
<<< direct Prolog Connection >>>
^Z
Stopped
path02: DelNet: BG
BG: not found
path02: DelNet: bg
[1]  prolog &
path02: DelNet:
/usr/users/csk/.finish: not found
Connection closed

delphi: init: rlogin duke
Last login: Sat Sep 16 16:23:30 from delphi
Ultrix V1.2 System #39: Fri Sep 1 11:22:32 BST 1989
Welcome to duke.

Terminal type = xterm
duke: csk: cd DelNet
duke: DelNet: prolog

init_connection_info: service for direct Prolog connection? test2
<<< direct Prolog Connection >>>
^Z
Stopped
duke: DelNet: bg
[1]  prolog &
duke: DelNet:
/usr/users/csk/.finish: not found
Connection closed

delphi: init: rlogin fylde
Last login: Sat Sep 16 16:24:22 from delphi
Ultrix V1.2 System #39: Fri Sep 1 11:22:32 BST 1989
Welcome to fylde.

Terminal type = xterm
fylde: csk: cd DelNet
fylde: DelNet: prolog

init_connection_info: service for direct Prolog connection? test0
<<< direct Prolog Connection >>>
^Z
Stopped
fylde: DelNet: bg
[1]  prolog &
fylde: DelNet:
```

```
/usr/users/csk/.finish: not found
Connection closed

delphi: init: rlogin hendy
Last login: Sat Sep 16 16:25:16 from fylde
Ultrix V1.2 System #39: Fri Sep 1 11:22:32 BST 1989
Welcome to hendy.

Terminal type = xterm
hendy: csk: cd DelNet
hendy: DelNet: prolog

init_connection_info: service for direct Prolog connection? test0
<<< direct Prolog Connection >>>
^Z
Stopped
hendy: DelNet: bg
[1]  prolog &
hendy: DelNet: prolog

init_connection_info: service for direct Prolog connection? test1
<<< direct Prolog Connection >>>
^Z
Stopped
hendy: DelNet: bg
[2]  prolog &
hendy: DelNet:
/usr/users/csk/.finish: not found
Connection closed

delphi: init: Controller

        TYPE AN 'h' FOR HELP

                top-level command ? h

        d       (d)irect connection to a prolog <USE WITH CAUTION>
        h       (h)elp, print this menu
        l       (l)ist current masters
        m       kill all (m)asters and their prologs
        p       kill all (p)rologs
        q       kill everything and (q)uit
        s       (s)tart or resume processing
        t       (t)alk to a host


                top-level command ? d

         hostname for direct connect to prolog? fylde

         service for direct connect to prolog? test0

        TYPE AN 'h' FOR HELP

                top-level command ? d

         hostname for direct connect to prolog? duke

         service for direct connect to prolog? test2

        TYPE AN 'h' FOR HELP

                top-level command ? d

         hostname for direct connect to prolog? path02

         service for direct connect to prolog? test0

        TYPE AN 'h' FOR HELP

                top-level command ? d

         hostname for direct connect to prolog? path01

         service for direct connect to prolog? test0

        TYPE AN 'h' FOR HELP
```

```
                top-level command ? d

         hostname for direct connect to prolog? path01

         service for direct connect to prolog? test1

        TYPE AN 'h' FOR HELP

                top-level command ? d

         hostname for direct connect to prolog? path01

         service for direct connect to prolog? test3

        TYPE AN 'h' FOR HELP

                top-level command ? d

         hostname for direct connect to prolog? hendy

         service for direct connect to prolog? test0

        TYPE AN 'h' FOR HELP

                top-level command ? d

         hostname for direct connect to prolog? hendy

         service for direct connect to prolog? test1

        TYPE AN 'h' FOR HELP

                top-level command ? h

        d       (d)irect connection to a prolog <USE WITH CAUTION>
        h       (h)elp, print this menu
        l       (l)ist current masters
        m       kill all (m)asters and their prologs
        p       kill all (p)rologs
        q       kill everything and (q)uit
        s       (s)tart or resume processing
        t       (t)alk to a host

                top-level command ? s

answer_sock: initialprologs = 8

PROBLEM HAS BEEN SOLVED

                top-level command ? h

        d       (d)irect connection to a prolog <USE WITH CAUTION>
        h       (h)elp, print this menu
        l       (l)ist current masters
        m       kill all (m)asters and their prologs
        p       kill all (p)rologs
        q       kill everything and (q)uit
        s       (s)tart or resume processing
        t       (t)alk to a host

                top-level command ? q

delphi: init: cat Global.Log

Sat Sep 16 16:56:03 1989  fylde  prolog_same: accepted client connection on fd 5 Socket 5

Sat Sep 16 16:56:17 1989  duke  prolog_same: accepted client connection on fd 5 Socket 6

Sat Sep 16 16:56:26 1989  path02  prolog_same: accepted client connection on fd 5 Socket 7

Sat Sep 16 16:56:34 1989  path01  prolog_same: accepted client connection on fd 5 Socket 8

Sat Sep 16 16:56:44 1989  path01  prolog_same: accepted client connection on fd 5 Socket 9
Sat Sep 16 16:56:51 1989  path01  prolog_same: accepted client connection on fd 5 Socket 10

Sat Sep 16 16:57:03 1989  hendy  prolog_same: accepted client connection on fd 5 Socket 11

Sat Sep 16 16:57:11 1989  hendy  prolog_same: accepted client connection on fd 5 Socket 12

Sat Sep 16 16:57:23 1989  Controller on host delphi, answer_sock: START TIMING

Sat Sep 16 16:57:29 1989  hendy  ANSWER  X =
[square(4,2),square(3,4),square(2,1),square(1,3)] Prolog socket 12

Sat Sep 16 16:57:37 1989  path01  ANSWER  X =
[square(4,3),square(3,1),square(2,4),square(1,2)] Prolog socket 10

Sat Sep 16 16:57:37 1989  Controller  PROBLEM HAS BEEN SOLVED time 13.630

Sat Sep 16 16:57:37 1989  Controller  Total Number of Checkins = 0
```

6b.2 Starting Master Processes

Starting Master Processes is an easier initialisation technique for the user. The user must still logon to each of the host machines individually, but only one process, the Prolog Master Server (PMasterSv), needs to be started on each host. The same service name (test0) has been used on all of the host machines for convenience. The Prolog Master Server waits on this port until a connection to the Controller is established.

The Controller is run with no command line arguments, so the interactive mode is entered. A single connection is made to the Master process running on each of the participating host machines. After this connection has been made, each Prolog can be started simply by issuing a command to the proper Master.

```
delphi: init: rlogin path01
Last login: Sat Sep 16 16:51:41 from delphi
Ultrix V2.2 System #285: Tue Sep  5 16:52:32 BST 1989

                Digital Equipment Corporation
                Merrimack, New Hampshire.

path01: csk: cd DelNet
path01: DelNet: PMasterSv
Usage:  PMasterSv <service>

path01: DelNet: PMasterSv test0
path01: DelNet: sps
Ty User      Status Proc# Command
p0.csk       RTTYP0  2455 -msh
p1 csk       CHILD   2591 -msh
p1. *        RUN     2615 sps
p1 csk       SOCKET  2614 PMasterSv test0
path01: DelNet:
/usr/users/csk/.finish: not found
Connection closed

delphi: init: rlogin path02
Last login: Sat Sep 16 16:52:44 from delphi
Ultrix-32 V3.0 (Rev 63) System #7: Wed Aug 16 17:28:04 BST 1989
Welcome to path02.

**** This system is running testing auto NFS mount code (16). pb 24 Aug 89 ****
```

```
Terminal type = xterm
path02: csk: cd DelNet
path02: DelNet: PMasterSv test0
path02: DelNet:
/usr/users/csk/.finish: not found
Connection closed

delphi: init: rlogin duke
Last login: Sat Sep 16 16:53:36 from delphi
Ultrix V1.2 System #39: Fri Sep 1 11:22:32 BST 1989
Welcome to duke.

Terminal type = xterm
duke: csk: cd DelNet
duke: DelNet: PMasterSv test0
duke: DelNet:
/usr/users/csk/.finish: not found
Connection closed

delphi: init: rlogin fylde
Last login: Sat Sep 16 16:54:30 from delphi
Ultrix V1.2 System #39: Fri Sep 1 11:22:32 BST 1989
Welcome to fylde.

Terminal type = xterm
fylde: csk:  cd DelNet
fylde: DelNet: PMasterSv test0
fylde: DelNet:
/usr/users/csk/.finish: not found
Connection closed

delphi: init: rlogin hendy
Last login: Sat Sep 16 16:55:14 from delphi
Ultrix V1.2 System #39: Fri Sep 1 11:22:32 BST 1989
Welcome to hendy.

Terminal type = xterm
hendy: csk: cd DelNet
hendy: DelNet: PMasterSv test0
hendy: DelNet:
/usr/users/csk/.finish: not found
Connection closed

delphi: init: Controller

        TYPE AN 'h' FOR HELP

                top-level command ? h

        d       (d)irect connection to a prolog <USE WITH CAUTION>
        h       (h)elp, print this menu
        l       (l)ist current masters
        m       kill all (m)asters and their prologs
        p       kill all (p)rologs
        q       kill everything and (q)uit
        s       (s)tart or resume processing
        t       (t)alk to a host


                top-level command ? l

                        LIST OF HOSTS AND CONNECTIONS

                there are no masters in the list

                top-level command ? t

                hostname? hendy

                service for host hendy? test0

        connected to host hendy,  TYPE AN 'h' FOR HELP

                        command to prolog master? h

        ON HOST hendy
```

```
c       kill all (c)hildren
h       (h)elp, print this menu
l       (l)ist children's process ids <AT THE SERVER SIDE>
k       (k)ill children and master then return to top level
q       (q)uit this loop and return to top level
s       (s)tart a prolog

                command to prolog master? s

        prolog started on host hendy

                command to prolog master? s

        prolog started on host hendy

                command to prolog master? q

        top-level command ? l

                LIST OF HOSTS AND CONNECTIONS

           hostname = hendy, socket = 5, number of prologs = 2
              children's sockets =   6, 7,

        top-level command ? t

        hostname? fylde

        service for host fylde? test0

connected to host fylde,  TYPE AN 'h' FOR HELP

                command to prolog master? s

        prolog started on host fylde

                command to prolog master? q

        top-level command ? path01

        top-level command ? t

        hostname? path01

        service for host path01? test0

connected to host path01,  TYPE AN 'h' FOR HELP

                command to prolog master? s

        prolog started on host path01

                command to prolog master? q

        top-level command ? l

                LIST OF HOSTS AND CONNECTIONS

           hostname = hendy, socket = 5, number of prologs = 2
              children's sockets =   6, 7,

           hostname = fylde, socket = 8, number of prologs = 1
              children's sockets =   9,

           hostname = path01, socket = 10, number of prologs = 1
              children's sockets =   11,

        top-level command ? t

        hostname? path02

        service for host path02? test0

connected to host path02,  TYPE AN 'h' FOR HELP

                command to prolog master? s
        prolog started on host path02

                command to prolog master? q

        top-level command ? l

                LIST OF HOSTS AND CONNECTIONS

           hostname = hendy, socket = 5, number of prologs = 2
              children's sockets =   6, 7,

           hostname = fylde, socket = 8, number of prologs = 1
              children's sockets =   9,

           hostname = path01, socket = 10, number of prologs = 1
              children's sockets =   11,

           hostname = path02, socket = 12, number of prologs = 1
              children's sockets =   13,

        top-level command ? t

        hostname? duke

        service for host duke? test0

connected to host duke,  TYPE AN 'h' FOR HELP

                command to prolog master? s

        prolog started on host duke

                command to prolog master? q

        top-level command ? t

        hostname? path01

connected to host path01,  TYPE AN 'h' FOR HELP

                command to prolog master? s

        prolog started on host path01

                command to prolog master? s

        prolog started on host path01

                command to prolog master? q

        top-level command ? l

                LIST OF HOSTS AND CONNECTIONS

           hostname = hendy, socket = 5, number of prologs = 2
              children's sockets =   6, 7,

           hostname = fylde, socket = 8, number of prologs = 1
              children's sockets =   9,

           hostname = path01, socket = 10, number of prologs = 3
              children's sockets =   11, 16, 17,

           hostname = path02, socket = 12, number of prologs = 1
              children's sockets =   13,

           hostname = duke, socket = 14, number of prologs = 1
              children's sockets =   15,

        top-level command ? h

d       (d)irect connection to a prolog <USE WITH CAUTION>
h       (h)elp, print this menu
l       (l)ist current masters
m       kill all (m)asters and their prologs
p       kill all (p)rologs
q       kill everything and (q)uit
```

```
        s       (s)tart or resume processing
        t       (t)alk to a host

                top-level command ? s
answer_sock: initialprologs = 8

PROBLEM HAS BEEN SOLVED

                top-level command ? q

delphi: init: cat Global.Log

Sat Sep 16 17:04:11 1989  hendy  prolog_same: accepted client connection on fd 3 Socket 6

Sat Sep 16 17:04:14 1989  hendy  prolog_same: accepted client connection on fd 3 Socket 7

Sat Sep 16 17:04:39 1989  fylde  prolog_same: accepted client connection on fd 3 Socket 9

Sat Sep 16 17:05:20 1989  path01  prolog_same: accepted client connection on fd 3 Socket 11

Sat Sep 16 17:06:05 1989  path02  prolog_same: accepted client connection on fd 3 Socket 13

Sat Sep 16 17:06:32 1989  duke  prolog_same: accepted client connection on fd 3 Socket 15

Sat Sep 16 17:06:42 1989  path01  prolog_same: accepted client connection on fd 3 Socket 16

Sat Sep 16 17:06:45 1989  path01  prolog_same: accepted client connection on fd 3 Socket 17

Sat Sep 16 17:07:10 1989  Controller on host delphi, answer_sock: START TIMING

Sat Sep 16 17:07:17 1989  hendy  ANSWER  X =
[square(4,2),square(3,4),square(2,1),square(1,3)] Prolog socket 7

Sat Sep 16 17:07:24 1989  path01  ANSWER  X =
[square(4,3),square(3,1),square(2,4),square(1,2)] Prolog socket 11

Sat Sep 16 17:07:24 1989  Controller  PROBLEM HAS BEEN SOLVED time 14.140

Sat Sep 16 17:07:24 1989  Controller  Total Number of Checkins = 0
```

6b.3 Automatic Start Up

With this method of start up the Controller process does not have to be run interactively. The configuration for the Delphi run is read from a file and no menu driven program is entered. The configuration file used in the automatic start up was:

```
path01      3
path02
duke
fylde
hendy       2
hythe
tholos
```

For this session, the `Controller` command is invoked with a single argument on the command line. This argument is the number of host machines to be used in the configuration. the lines of the configuration file are read in order until that number of available host can be found and a connection established. Once this command has been given, there is no more user interaction with the Controller process. In this session the machine `path02` is purposely crashed during the initialisation procedure. The Controller automatically reads the next line in the configuration file, and starts the requested number of Prologs (1) on the new host machines(`hythe`).

A second argument on the commands line is allowed, but has not been used in any of these sessions. The second argument is the name of the file to be used as the standard input to the Prolog processes. The full initiation command is:

`Controller` <number of hosts> <file to be used as standard input>

The default filename is `prolog_standard_input`. Allowing an arbitrary file to be read is useful if multiple problems are being executed by the same user, in separate configurations.

```
delphi: init: Controller 5

                3 prolog(s) have been started on host path01

        Daemon not running on host path02
        could not connect to hostname path02 and start prologs...machine crashed recently?

                1 prolog(s) have been started on host duke

                1 prolog(s) have been started on host fylde

                2 prolog(s) have been started on host hendy

                1 prolog(s) have been started on host hythe

answer_sock: initialprologs = 8

PROBLEM HAS BEEN SOLVED
delphi: init:
delphi: init: cat Global.Log

Sat Sep 16 17:08:14 1989  path01  prolog_same: accepted client connection on fd 3 Socket 7

Sat Sep 16 17:08:14 1989  path01  prolog_same: accepted client connection on fd 3 Socket 8

Sat Sep 16 17:08:15 1989  path01  prolog_same: accepted client connection on fd 3 Socket 9

Sat Sep 16 17:08:17 1989  duke  prolog_same: accepted client connection on fd 3 Socket 11

Sat Sep 16 17:08:20 1989  fylde  prolog_same: accepted client connection on fd 3 Socket 13

Sat Sep 16 17:08:23 1989  hendy  prolog_same: accepted client connection on fd 3 Socket 15

Sat Sep 16 17:08:23 1989  hendy  prolog_same: accepted client connection on fd 3 Socket 16

Sat Sep 16 17:08:25 1989  hythe  prolog_same: accepted client connection on fd 3 Socket 18

Sat Sep 16 17:08:32 1989  Controller on host delphi, answer_sock: START TIMING

Sat Sep 16 17:08:39 1989  hendy  ANSWER  X =
[square(4,2),square(3,4),square(2,1),square(1,3)] Prolog socket 15

Sat Sep 16 17:08:46 1989  path01  ANSWER  X =
[square(4,3),square(3,1),square(2,4),square(1,2)] Prolog socket 7

Sat Sep 16 17:08:46 1989  Controller  PROBLEM HAS BEEN SOLVED time 13.980

Sat Sep 16 17:08:46 1989  Controller  Total Number of Checkins = 0
```

Appendix 7a Encodings

7a.1 Regular and Compressed Encodings

Generating regular and compressed encodings can be seen as different methods for locally (for each set of clauses) balancing binary trees. To be balanced each vertex in the tree must have left and right subtrees which differ in depth by at most one. An additional property of the compressed encoding is that a strictly binary tree is formed. A strictly binary tree is a tree (inward branching factor equal to one) which has vertices with an outward branching factor equal to zero or two.

With a regular encoding each of the n objects is allocated the same number of bits. This means that all leaf nodes of the binary tree will be located at the same level. Each object (clause in the set) is encoded with $\lceil \log_2 n \rceil$ bits giving an overall total of $n\lceil \log_2 n \rceil$ bits for all n encodings.

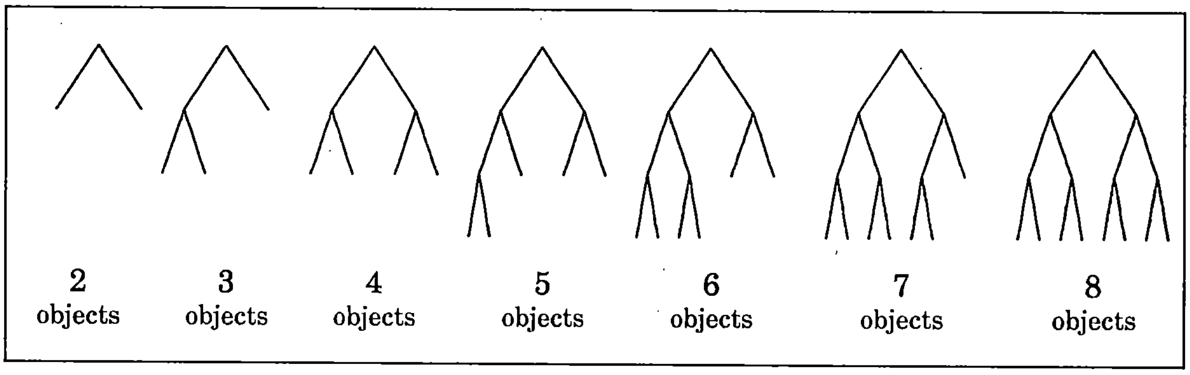


Figure 7a.1 CompressedEncodings

Compressed encodings form strictly binary trees. The leaf nodes of these trees must be within one level of each other. The maximum number of bits needed to encode n objects is $\lceil \log_2 n \rceil$ for some portion of those objects and $\lceil \log_2 n \rceil - 1$ for others. Figure 7a.1 graphically shows compressed encodings for from two through eight objects. Each encoding of n objects contains $\lceil \log_2 n \rceil + 1$ more bits than the previous encoding (n – 1 objects). This gives the following recurrence relation for the total number of bits needed to encode n objects:

$$T_2 = 2$$
$$T_n = T_{n-1} + \lceil \log_2 n \rceil + 1 \quad ; n > 2 \qquad [*]$$

To obtain a closed form, we observe that some number of the n encodings are length $\lceil \log_2 n \rceil$ and some are of length $\lceil \log_2 n \rceil - 1$ giving an overall count:

$$C_n = N\lceil \log_2 n \rceil + M(\lceil \log_2 n \rceil - 1)$$

7a.2 Induction Proof of C_n

M and N in equation $C_n = N\lceil \log_2 n\rceil + M(\lceil \log_2 n\rceil - 1)$, are found to have the values:

$$M = 2^{\lceil \log_2 n\rceil} - n$$

and

$$N = n - M$$

using q to represent $\lceil \log_2 n\rceil$:

$$C_n = (n - M)q + M(q - 1)$$
$$= (n - 2^q + n)q + (2^q - n)(q - 1)$$
$$= 2nq - q2^q + q2^q - 2^q - nq + n$$
$$C_n = n(q + 1) - 2^q \qquad [\dagger]$$

A proof by induction that C_n is a closed form of T_n follows:

$$C_2 = 2(\lceil \log_2 2\rceil + 1) - 2^{\lceil \log_2 2\rceil}$$
$$C_2 = 2 = T_2$$

$$\text{Assume} \quad C_{n-1} = (n-1)(\lceil \log_2 (n-1)\rceil + 1) - 2^{\lceil \log_2 (n-1)\rceil}$$
$$\text{So,} \quad T_n = (n-1)(\lceil \log_2 (n-1)\rceil + 1) - 2^{\lceil \log_2 (n-1)\rceil} + \lceil \log_2 n\rceil + 1$$

There are two cases to consider:

I when $n \neq 2^m + 1$

II when $n = 2^m + 1$

By definition of T_n, $n > 2$ (see [*]) so $m > 0$

I For the trivial case where $n \neq 2^m + 1$, $\lceil \log_2 n\rceil = \lceil \log_2 (n-1)\rceil$.

$$\text{Using } r = \lceil \log_2 n\rceil = \lceil \log_2 (n-1)\rceil$$
$$T_n = (n-1)(r + 1) - 2^r + r + 1$$
$$= nr + n - r - 1 - 2^r + r + 1$$
$$T_n = n(r + 1) - 2^r \text{ which is } C_n$$

II For the case where $n = 2^m + 1$, $\lceil \log_2 n\rceil = \lceil \log_2 (n-1)\rceil + 1$.

$$T_n = (n-1)(\lceil \log_2 (n-1)\rceil + 1) - 2^{\lceil \log_2 (n-1)\rceil} + \lceil \log_2 n\rceil + 1$$
$$= n\lceil \log_2 (n-1)\rceil + n - \lceil \log_2 (n-1)\rceil - 1 - 2^{\lceil \log_2 (n-1)\rceil} + \lceil \log_2 n\rceil + 1$$

Since $n-1 = 2^m$, $2^{\lceil \log_2(n-1) \rceil} = 2^{\log_2(n-1)} = n-1$

$$T_n = n(\lceil \log_2 n \rceil - 1) + n - (\lceil \log_2 n \rceil - 1) - 1 - (n-1) + \lceil \log_2 n \rceil + 1$$
$$= n\lceil \log_2 n \rceil - n + n - \lceil \log_2 n \rceil + 1 - 1 - n + 1 + \lceil \log_2 n \rceil + 1$$

Rearranging:

$$T_n = n\lceil \log_2 n \rceil + n - n + 1 - n + 1 - \lceil \log_2 n \rceil + \lceil \log_2 n \rceil - 1 + 1$$

Another formulation of $\lceil \log_2 n \rceil = \lceil \log_2(n-1) \rceil + 1$ gives:

$$2^{\lceil \log_2 n \rceil} = 2^{\lceil \log_2(n-1) \rceil + 1} = 2^{\lceil \log_2(n-1) \rceil} \cdot 2$$

Since $n-1 = 2^{\lceil \log_2(n-1) \rceil}$, $2(n-1) = 2^{\lceil \log_2 n \rceil}$ and:

$$T_n = n\lceil \log_2 n \rceil + n - 2(n-1) - \lceil \log_2 n \rceil + \lceil \log_2 n \rceil - 1 + 1$$
$$T_n = n(\lceil \log_2 n \rceil + 1) - 2^{\lceil \log_2 n \rceil}$$

which again is shown to be C_n and completes the proof.

7a.3 Aside

If instead of using a ceiling function as in C_n (see [†]), we had used the floor:

using p to represent $\lfloor \log_2 n \rfloor$:

the formula for the number of bits needed to encode n objects would be:

$$F_n = n(p+2) - 2^{p+1}$$

Equating these two expressions results in an elegant identity between the floor and ceiling:

$$n(p+2) - 2^{p+1} = n(q+1) - 2^q$$
$$n(p+1) - 2^{p+1} = nq - 2^q$$
$$n(p+1-q) = 2^{p+1} - 2^q$$

Appendix 8a All Check-ins

This appendix contains the execution times for numerous runs of the 8-queens problem. The reassign jobs control strategy was used with a range of check-in intervals from one through an interval greater than the depth of the 8-queens search space.

Table 8a.1 through Table 8a.8 are the results from setting the initial check-in interval to the values one through eighty. The numbers from one to twenty in the left-hand column are the number of participating host machines. The numbers within the table are execution times in seconds for the particular configuration. A zero in the table represents a timing that was unobtainable for any of the following reasons:

- Not enough host machines were up.
- A Prolog process was killed during execution.
- A host machine crashed (or was rebooted) during processing.

There are fault tolerant mechanisms built into Delphi to handle process or machine crashes. These were turned off during the benchmarking of this problem. Two reasons are given for this decision. The first is that the timings should not include the restart of Prolog processes when machines are rebooted. If a machine has crashed then Delphi will start up a Prolog process on an alternative host machine. The second reason is a matter of network etiquette. Many of these timings took place during the working hours of other network users. The major reason that a Prolog process crashes is that a machine has been rebooted by another user. Apart from this Prologs only die when a user has specifically killed them. If the Prolog dies and the machine it was running on is still functioning, Delphi automatically restarts the Prolog on the same host machine. This can be very annoying to other users who have purposely killed the Prolog processes to run their own.

Table 8a.9 shows check-in intervals which are much higher than the maximum depth of the 8-queens search space. What is being demonstrated in this appendix is that control communications or check-ins are not needed in the 8-queens (nor any of the N-queens) problem. The optimal results in terms of a low execution time are obtained even when there are no control communications performed at all.

8a.1 Check-in Intervals 1 Through 80

	1	2	3	4	5	6	7	8	9	10
1	8321	3970	2416	1769	1419	1225	1082	955	828	723
2	4217	2041	1268	971	771	683	619	544	463	405
3	2852	1395	865	636	553	475	417	365	335	288
4	2682	1127	683	501	410	377	310	289	252	223
5	2705	1059	569	427	356	298	276	236	209	205
6	2698	1056	560	381	292	247	234	211	177	157
7	2705	1070	558	368	276	233	212	195	163	165
8	2720	1095	586	380	276	222	187	171	151	140
9	2720	1094	593	386	281	227	193	169	149	122
10	2730	1111	605	398	294	229	203	173	147	125
11	2754	1110	601	395	304	251	209	172	151	132
12	2756	1129	604	408	296	248	199	180	149	119
13	2774	1147	619	405	304	252	217	176	168	133
14	2782	1132	610	426	313	249	220	196	155	126
15	2784	0	605	440	315	271	221	180	174	135
16	2799	1147	625	419	328	250	227	200	170	138
17	2822	1188	636	446	331	266	233	196	181	143
18	2816	1144	634	411	323	267	232	202	170	157
19	2829	1196	644	420	324	271	230	194	171	165
20	2840	1163	638	415	323	269	249	205	190	160

Table 8a.1 Check-in Intervals 1 Through 10

	11	12	13	14	15	16	17	18	19	20
1	655	598	565	537	507	479	454	435	426	405
2	381	341	315	310	309	272	304	297	269	255
3	273	248	228	207	241	189	222	189	234	191
4	222	212	197	173	174	190	181	224	180	196
5	193	162	164	148	160	157	138	147	158	165
6	146	140	129	130	130	130	127	136	127	143
7	129	133	123	121	123	104	132	112	114	100
8	121	111	105	103	95	101	102	81	88	79
9	110	100	98	91	89	81	77	76	69	62
10	113	103	98	88	85	82	67	69	70	63
11	114	112	96	87	83	87	74	68	68	65
12	114	98	96	94	86	83	79	71	74	61
13	123	103	95	93	93	76	78	73	67	63
14	122	116	111	88	86	92	69	65	68	63
15	122	116	108	95	94	90	77	63	68	66
16	130	116	106	91	103	86	79	71	70	60
17	139	120	112	100	99	90	78	79	69	68
18	0	130	106	104	103	86	83	69	68	58
19	0	114	110	105	96	92	94	73	88	67
20	0	133	107	101	109	95	89	76	68	66

Table 8a.2 Check-in Intervals 11 Through 20

	21	22	23	24	25	26	27	28	29	30
1	398	388	381	377	369	358	353	345	339	329
2	256	250	236	221	224	219	212	191	201	188
3	172	163	180	237	186	176	192	250	187	208
4	198	142	238	197	179	206	129	209	212	241
5	182	156	122	149	142	205	258	0	188	128
6	116	130	127	156	120	129	151	0	159	104
7	97	0	90	106	123	120	163	111	86	111
8	82	0	96	72	96	114	75	0	80	70
9	64	0	63	65	74	79	64	0	53	54
10	62	0	61	59	67	57	59	0	54	48
11	64	0	62	62	62	57	63	0	51	41
12	62	0	58	62	50	59	51	67	46	46
13	58	0	66	53	55	49	50	61	50	55
14	61	0	57	60	52	53	48	0	43	43
15	66	0	53	56	53	52	54	0	44	51
16	63	0	56	55	45	53	55	0	45	38
17	61	0	60	55	53	51	49	0	42	51
18	69	0	63	60	51	50	50	0	48	61
19	67	0	66	55	53	50	47	0	44	42
20	58	0	0	0	0	0	0	0	0	0

Table 8a.3 Check-in Intervals 21 Through 30

	31	32	33	34	35	36	37	38	39	40
1	323	314	308	304	304	297	293	296	293	293
2	180	177	181	202	175	177	276	295	293	294
3	194	174	155	178	189	220	469	547	551	543
4	143	149	135	180	174	323	477	479	478	478
5	112	91	122	119	193	355	395	395	387	398
6	132	133	120	95	210	300	306	303	301	311
7	100	132	77	138	185	217	215	214	216	220
8	70	58	106	98	126	132	134	132	132	132
9	50	64	47	55	61	60	60	60	61	60
10	47	40	62	73	46	44	44	43	43	44
11	71	47	64	46	45	45	45	43	44	47
12	43	42	48	45	43	44	44	45	48	45
13	46	43	43	73	44	43	44	46	53	44
14	62	53	45	46	45	43	44	44	45	50
15	46	54	53	44	44	45	47	47	44	43
16	40	43	53	45	45	44	44	44	45	44
17	44	37	47	36	45	43	45	43	48	45
18	42	41	41	37	44	45	43	45	46	45
19	46	41	36	36	45	44	43	43	53	45
20	0	0	0	0	0	0	0	0	0	0

Table 8a.4 Check-in Intervals 31 Through 40

	41	42	43	44	45	46	47	48	49	50
1	292	294	293	293	297	292	292	292	292	292
2	294	299	293	294	294	293	293	292	292	292
3	556	558	553	554	547	558	553	556	552	548
4	480	478	484	460	477	483	477	481	478	478
5	394	393	393	410	281	346	392	395	393	392
6	361	171	222	272	316	272	303	305	305	302
7	214	216	117	215	278	142	183	215	174	215
8	133	87	118	92	139	133	119	131	104	136
9	60	76	74	76	88	81	60	63	76	62
10	85	76	57	76	74	64	82	47	72	53
11	87	77	68	76	80	61	46	54	44	48
12	50	51	44	55	72	47	60	45	43	56
13	48	60	60	47	80	44	48	47	45	44
14	44	51	67	46	52	43	44	44	43	44
15	50	50	43	44	45	63	52	44	58	43
16	54	50	45	49	57	44	48	43	43	46
17	46	46	45	69	0	0	46	0	44	73
18	46	45	43	49	0	0	45	0	43	44
19	44	50	44	0	0	0	43	60	43	44
20	0	0	0	0	0	0	0	0	0	0

Table 8a.5 Check-in Intervals 41 Through 50

	51	52	53	54	55	56	57	58	59	60
1	292	292	292	292	293	292	292	293	298	292
2	296	293	294	294	293	293	293	294	298	293
3	552	549	551	551	548	549	550	554	550	551
4	479	472	476	478	471	480	473	481	470	476
5	393	390	388	392	391	392	390	302	388	397
6	304	303	301	305	305	304	304	302	304	391
7	215	214	216	216	214	217	214	222	120	227
8	131	93	130	131	130	130	131	181	130	149
9	61	74	60	60	61	61	74	74	61	76
10	60	43	43	44	43	43	60	60	45	75
11	44	65	44	44	44	44	45	46	43	44
12	43	44	44	44	43	44	51	48	50	49
13	43	43	43	43	43	43	44	44	44	47
14	44	43	43	44	42	44	44	43	43	43
15	43	66	43	44	43	43	44	47	51	48
16	45	59	43	43	44	44	44	43	43	43
17	43	43	43	43	44	45	61	48	43	46
18	67	44	43	43	44	44	46	43	43	47
19	67	44	43	43	43	43	45	43	49	44
20	0	0	0	0	0	0	0	0	0	0

Table 8a.6 Check-in Intervals 51 Through 60

	61	62	63	64	65	66	67	68	69	70
1	292	293	292	293	293	292	293	292	293	292
2	294	294	293	295	293	293	293	296	293	293
3	552	546	550	547	546	543	546	543	548	545
4	477	478	476	472	474	472	473	474	476	471
5	389	389	390	397	386	387	392	389	391	391
6	217	302	298	304	300	306	302	302	301	305
7	214	214	213	214	214	218	111	216	214	214
8	129	130	86	87	131	131	131	132	131	131
9	61	61	61	60	61	61	60	61	60	60
10	61	43	60	44	43	43	44	44	45	43
11	52	44	44	43	44	44	44	43	44	43
12	45	44	69	0	43	44	43	44	44	43
13	46	43	44	70	43	44	43	44	43	44
14	51	44	43	43	43	43	44	43	44	43
15	51	53	43	42	44	44	43	43	43	43
16	52	44	60	43	43	43	43	43	44	43
17	44	43	43	43	43	48	43	43	44	43
18	43	43	47	44	43	48	43	43	43	44
19	45	44	48	43	43	44	43	43	44	44
20	0	0	0	0	0	0	0	0	0	0

Table 8a.7 Check-in Intervals 61 Through 70

	71	72	73	74	75	76	77	78	79	80
1	292	291	292	292	296	287	325	291	286	290
2	294	294	294	295	293	288	289	288	286	287
3	549	556	549	548	550	264	259	262	259	261
4	478	482	481	473	476	227	228	277	224	228
5	391	394	400	391	391	192	289	285	280	187
6	303	302	172	302	156	219	165	212	143	205
7	213	217	217	217	215	115	202	175	174	105
8	132	131	132	131	132	66	94	108	107	77
9	61	61	61	60	61	63	74	66	65	43
10	43	43	61	44	44	112	66	64	64	57
11	43	43	48	43	50	46	66	62	59	43
12	43	43	43	43	44	50	67	49	59	57
13	43	43	62	43	44	44	64	50	49	43
14	43	44	53	43	44	47	54	46	45	45
15	43	43	50	43	43	49	53	49	44	44
16	44	44	53	43	43	45	48	49	50	47
17	43	43	43	45	44	44	45	49	43	46
18	44	45	45	43	43	42	43	44	44	43
19	43	43	43	43	43	44	44	50	43	44
20	0	0	0	0	0	42	44	43	43	42

Table 8a.8 Check-in Intervals 71 Through 80

8a.2 Check-in Interval 81 and Selected Others

	81	82	83	90	100	1000	10000
1	289	286	289	287	296	288	288
2	288	289	286	287	289	286	288
3	261	261	261	262	262	264	259
4	227	230	226	229	225	232	228
5	213	176	171	197	187	206	192
6	170	160	164	182	141	172	165
7	110	115	110	96	101	127	102
8	61	62	64	46	62	72	64
9	49	62	49	42	43	63	54
10	50	51	49	44	44	41	51
11	44	41	44	45	42	44	44
12	44	44	42	45	43	43	45
13	42	42	41	45	41	44	44
14	42	41	44	42	42	44	42
15	43	42	42	42	41	44	41
16	42	42	44	42	43	42	41
17	42	43	41	41	42	41	41
18	42	43	43	42	43	41	43
19	43	42	41	43	42	41	44
20	42	41	42	42	43	42	41

Table 8a.9 Check-in Interval 81 and Selected Others

Appendix 8b Programs

8b.1 Prolog Top Level

The following code is the Prolog top level which each Path Processor runs. This code is shown to explain why no `write` statements are included in any of the source Prolog programs. The reason is that this top-level code automatically prints solutions both to the global log file (via an answer buffer) and to the local log file. The time taken to print all solutions to a query is included in the execution times shown within this dissertation.

```
/* This is the top-level read-eval-print loop for backtracking Delphi control strategies*/

mytop :-
        repeat,
        $run_type(Run), $newgoal(Run),
        read(X, Vars),
        $rerun_goal(Run, X, Vars).

$run_type(test).

$newgoal(test) :- writename('Newgoal Please ? -').

$do_proc(X,[]) :- !,$do_call(X),!,
        writename(yes),nl,fail.


/* The following is the second clause of do_proc if you want
   to allow backtracking to take place.*/
$do_proc(X,Vars) :- $do_call(X),
        $printans_toboth(Vars),fail.

/* $no_more_ans fails with any character except a line feed. When
   it fails, more answers are attempted. If it succeeds, no more
   answers are attempted.*/
$no_more_ans :- get0(10),!.
$no_more_ans :- $no_more_ans,fail.

$do_call(X) :- call(X).
$do_call(_) :- fail.


$rerun_goal(test, X, Vars) :- $do_proc(X,Vars).
$rerun_goal(test, X, Vars) :-
        orc_rerun_goal, !,
        $rerun_goal(test, X, Vars).

$printans_tobuf([]) :- done_answer_tobuf, !.
$printans_tobuf([=(Name, Val)|Tail]) :-
                writename_tobuf(' '), writename_tobuf(Name),
                writename_tobuf(' = '),
                write_tobuf(Val),
                $printans_tobuf(Tail), !.

$printans_toboth([]) :- done_answer_tobuf, !.
$printans_toboth([=(Name, Val)|Tail]) :-
                nl,writename_tobuf(' '),writename(Name),writename_tobuf(Name),
                writename(' = '),writename_tobuf(' = '),
                write(Val),write_tobuf(Val),
                $printans_toboth(Tail), !.
```

8b.2 N-Queens Source

The N-queens problem is to place N queens on an NxN chess board so that no two queens are attacking each other. The N-queens queries attempted were the following:

problem name	goal query	comments
2-queens	`get_solutions(2,X).`	This query was not used in benchmarking any implementation of the Delphi machine. It was used to test some of the tools associated with Delphi
3-queens	`get_solutions(3,X).`	This query was not used in benchmarking any implementation of the Delphi machine. It was used to test some of the tools associated with Delphi
4-queens	`get_solutions(4,X).`	This query was mainly used for testing the Amoeba-transactions-under-UNIX implementation
8-queens	`get_solutions(8,X).`	
9-queens	`get_solutions(9,X).`	
10-queens	`get_solutions(10,X).`	

```
get_solutions(Board_size, Soln) :- solve(Board_size, [], Soln).

solve(Bs, [square(Bs, Y) | L], [square(Bs, Y) | L]).
solve(Board_size, Initial, Final) :-
                 newsquare(Initial, Next, Board_size),
                 solve(Board_size, [Next | Initial], Final).

newsquare([square(I,J) | Rest], square(X, Y), Boardsize) :-
        I < Boardsize, X is I + 1, snint(Y, Boardsize),
        notthreatened(I, J, X, Y), safe(X, Y, Rest).
newsquare([], square(1, X), Boardsize) :- snint(X, Boardsize).

snint(X, X).
snint(N, NPlusOneOrMore) :- M is NPlusOneOrMore - 1, M > 0,
        snint(N,M).

notthreatened(I, J, X, Y) :- I \== X, J \== Y,
        U1 is I - J, V1 is X - Y, U1 \== V1,
        U2 is I + J, V2 is X + Y, U2 \== V2.

safe(X, Y, []).
safe(X, Y, [square(I, J) | L]) :-
        notthreatened(I, J, X, Y), safe(X, Y, L).
```

8b.3 Pentominoes Source

The pentominoes problem is a shape fitting problem. Twelve shapes have to be arranged on a board so that the board is covered and all shapes are used exactly once. The name comes from the fact that the shapes are made up of five identical squares. There are twelve unique ways to arrange the five squares. Each of these arrangements is called a pentomino. In this problem, the board is size 3x20. The source for this program was adapted from Lusk, Overbeek *et. al.* [1987]. The query for this problem was the following:

problem name	**goal query**
pentominoes	`solution(Soln).`

```
solution(H) :-
        initial_state(Si),
        can_reach(Si,Sf),
        final_state(Sf),
        Sf = state(_,_,H).

initial_state(state(Board,[1,2,3,4,5,6,7,8,9,10,11,12],[])) :-
        gen_board(20,Board).

gen_board(0,[]).
gen_board(N,[no_piece,no_piece,no_piece,border|T]) :-
        N > 0,
        I is (N - 1),
        gen_board(I,T).

final_state(state(_,[],_)).

can_reach(S1,S2) :-
        trans(S1,S),
        S = S2.
can_reach(S1,S2) :-
        trans(S1,S),
        can_reach(S,S2).

trans(State,New_State) :-
        State = state(Board,Pieces,History),
        delete(Piece,Pieces,New_Pieces),
        pent(Piece,Orientation,Pattern),
        play_pent(Board,Pattern,New_Board),
        New_State = state(New_Board,New_Pieces,
                [[Piece,Orientation] | History]).

delete(X,[X|Y],Y).
delete(X,[Y|Z], [Y|Z1]) :- delete(X,Z,Z1).

play_pent(Board,Pattern,New_Board) :-
        match(Board,Pattern,Board1),
        trim(Board1,New_Board).

trim([],[]).
trim([border|T],Board) :- trim(T,Board).
trim([piece|T],Board) :- trim(T,Board).
trim(Board,Board) :- Board = [no_piece|_].
```

```
match(Board,[],Board).
match([piece|Tb],[dnm|Tp],[piece|Tnb]) :-
        match(Tb,Tp,Tnb).
match([piece|Tb],[op|Tp],[piece|Tnb]) :-
        match(Tb,Tp,Tnb).
match([no_piece|Tb],[np|Tp],[piece|Tnb]) :-
        match(Tb,Tp,Tnb).
match([no_piece|Tb],[dnm|Tp],[no_piece|Tnb]) :-
        match(Tb,Tp,Tnb).
match([border|Tb],[dnm|Tp],[border|Tnb]) :-
        match(Tb,Tp,Tnb).


pent(1,1,[np,np,np,dnm,np,dnm,np]).
pent(1,2,[np,op,np,dnm,np,np,np]).
pent(1,3,[np,np,dnm,dnm,np,dnm,dnm,dnm,np,np]).
pent(1,4,[np,np,dnm,dnm,dnm,np,dnm,dnm,np,np]).

pent(2,1,[np,op,dnm,np,np,np,dnm,dnm,np]).

pent(3,1,[np,np,np,dnm,dnm,dnm,np,dnm,dnm,dnm,np]).
pent(3,2,[np,np,np,dnm,np,dnm,dnm,dnm,np]).
pent(3,3,[np,dnm,op,op,np,dnm,np,np,np]).
pent(3,4,[np,op,op,dnm,np,op,op,dnm,np,np,np]).

pent(4,1,[np,op,dnm,op,np,op,dnm,np,np,np]).
pent(4,2,[np,op,op,dnm,np,np,np,dnm,np]).
pent(4,3,[np,dnm,np,np,np,dnm,dnm,dnm,np]).
pent(4,4,[np,np,np,dnm,dnm,np,dnm,dnm,dnm,np]).

pent(5,1,[np,np,dnm,np,np,np]).
pent(5,2,[np,np,dnm,dnm,np,np,dnm,dnm,np]).
pent(5,3,[np,np,op,dnm,np,np,np]).
pent(5,4,[np,np,dnm,dnm,np,np,dnm,dnm,dnm,np]).
pent(5,5,[np,np,np,dnm,np,np]).
pent(5,6,[np,np,np,dnm,dnm,np,np]).
pent(5,7,[np,dnm,dnm,dnm,np,np,dnm,dnm,np,np]).
pent(5,8,[np,dnm,dnm,np,np,dnm,dnm,np,np]).

pent(6,1,[np,dnm,op,np,np,dnm,np,np]).
pent(6,2,[np,op,op,dnm,np,np,op,dnm,dnm,np,np]).
pent(6,3,[np,np,op,dnm,dnm,np,np,dnm,dnm,dnm,np]).
pent(6,4,[np,np,dnm,np,np,dnm,dnm,np]).

pent(7,1,[np,dnm,dnm,dnm,np,dnm,dnm,dnm,np,dnm,dnm,np,np]).
pent(7,2,[np,dnm,dnm,dnm,np,dnm,dnm,dnm,np,dnm,dnm,dnm,np,np]).
pent(7,3,[np,np,dnm,dnm,np,dnm,dnm,dnm,np,dnm,dnm,dnm,np]).
pent(7,4,[np,np,dnm,dnm,dnm,np,dnm,dnm,dnm,np,dnm,dnm,dnm,np]).

pent(8,1,[np,dnm,dnm,dnm,np,dnm,dnm,dnm,np,np,dnm,dnm,np]).
pent(8,2,[np,dnm,dnm,dnm,np,np,dnm,dnm,np,dnm,dnm,dnm,np]).
pent(8,3,[np,dnm,dnm,dnm,np,dnm,dnm,np,np,dnm,dnm,dnm,np]).
pent(8,4,[np,dnm,dnm,np,np,dnm,dnm,dnm,np,dnm,dnm,dnm,np]).

pent(9,1,[np,np,op,dnm,dnm,np,np,dnm,dnm,np]).
pent(9,2,[np,dnm,np,np,np,dnm,dnm,np]).
pent(9,3,[np,op,op,dnm,np,np,np,dnm,dnm,np]).
pent(9,4,[np,np,dnm,np,np,dnm,dnm,dnm,np]).
pent(9,5,[np,op,dnm,np,np,op,dnm,dnm,np,np]).
pent(9,6,[np,op,dnm,op,np,np,dnm,np,np]).
pent(9,7,[np,op,dnm,np,np,np,dnm,dnm,dnm,np]).
pent(9,8,[np,op,dnm,np,np,np,dnm,np]).

pent(10,1,[np,np,dnm,op,np,dnm,dnm,np,np]).
pent(10,2,[np,np,op,dnm,dnm,np,op,dnm,dnm,np,np]).
pent(10,3,[np,op,op,dnm,np,np,np,dnm,dnm,dnm,np]).
pent(10,4,[np,dnm,np,np,np,dnm,np]).

pent(11,1,[np,dnm,dnm,dnm,np,dnm,dnm,np,np,dnm,dnm,np]).
pent(11,2,[np,dnm,dnm,dnm,np,dnm,dnm,dnm,np,np,dnm,dnm,dnm,np]).
pent(11,3,[np,dnm,dnm,dnm,np,np,dnm,dnm,dnm,np,dnm,dnm,dnm,np]).
pent(11,4,[np,dnm,dnm,np,np,dnm,dnm,np,dnm,dnm,dnm,np]).

pent(12,1,[np,dnm,dnm,dnm,np,dnm,dnm,dnm,np,dnm,dnm,dnm,np,dnm,dnm,dnm,np]).
```

8b.4 Parser Source

The parser problem is a simplified version of a program written by Bob Moore of SRI International. It was adapted for use in this research by the authors of Alshawi and Moran [1988] of SRI International at the Cambridge Computer Science Research Centre, Cambridge, England. This problem is similar to the one investigated by Alshawi and Moran in their Delphi research [1988]. The problem is to parse natural language sentences presented as a list of words. For a more detailed description of the parser see Alshawi and Moran [1988]. The parser queries attempted were the following:

problem name	goal query	comments
parser-2	`parse([wren,built,a,chapel, in,a,college, in,a,college], Output).`	The parser problems are named by the number of `in,a,college` sequences contained in the query.
parser-3	`parse([wren,built,a,chapel, in,a,college, in,a,college, in,a,college], Output).`	The parser-3 query includes three `in,a,college` sequences.
parser-4	`parse([wren,built,a,chapel, in,a,college, in,a,college, in,a,college, in,a,college], Output).`	The parser-4 query includes four `in,a,college` sequences.

```
% start

parse(Input,StartCat) :-
     start_cat(StartCat),
     shift_reduce(sigma,[StartCat],[],Input).

% done

shift_reduce(sigma,[],[],[]).

% reduce

shift_reduce(Goal,[],stack(Goal1,First1,Rest1,Stack1),Input) :-
     predict_match(Goal,Goal1,First1,Rest1,Stack1,Input).
```

```
% shift

shift_reduce(Goal,[ListFirst|ListRest],Stack,[InputFirst|InputRest]) :-
     cat(InputFirst,Constituent),
  %N.B.  removed from parser
  %%%can_start(Constituent,ListFirst),
     predict_match(Constituent,Goal,ListFirst,ListRest,Stack,InputRest).

% shift gap

shift_reduce(Goal,[ListFirst|ListRest],Stack,Input) :-
     gap_can_start(ListFirst,Constituent),
     predict_match(Constituent,Goal,ListFirst,ListRest,Stack,Input).

% match

predict_match(Constituent,Goal,Constituent,Rest,Stack,Input) :-
     shift_reduce(Goal,Rest,Stack,Input).

% predict

predict_match(Constituent,Goal,First,Rest,Stack,Input) :-
     rule_can_start(Constituent,First,Goal1,List1),
     shift_reduce(Goal1,List1,stack(Goal,First,Rest,Stack),Input).

rule_can_start(Constituent,First,Goal1,List1) :-
     rule(Goal1,[Constituent|List1]),
     can_start(Goal1,First).


%-----------------------------------------------


start_cat(s(syn(tnsd,[],[],_,[],[]),_)).


can_start(vp(A,B),vp(C,D)).
can_start(vp(syn(A,ing,B,C,y,D,n,E,F,G),H),pred(syn(I,J,K,L,M,N,O,P),Q)).
can_start(vp(syn(A,passive,B,C,y,D,y,E,F,G),H),pred(syn(I,J,K,L,M,N,O,P),Q)).
can_start(np(A,B),np(C,D)).
can_start(np(syn(A,[],[],B,C,n,D,[],[],n),E),s(syn(tnsd,F,G,decl,H,I),J)).
can_start(np(syn(A,[],[],B,C,n,D,[],[],n),E),rel(syn(F),G)).
can_start(np(syn(A,[],[],B,C,n,D,[],[],q),E),s(syn(tnsd,F,F,whq,G,G),H)).
can_start(np(syn(A,[],[],B,C,n,D,[],[],r),E),rel(syn(F),G)).
can_start(s(A,B),s(C,D)).
can_start(s(syn(tnsd,[np(syn(A,[],[],B,C,n,y,[],[],n),D)],[],decl,[],[g]),E),rel(syn(F),G)).
can_start(aux(A,B),aux(C,D)).
can_start(aux(syn(A,tnsd,B,C,D),E),s(syn(tnsd,F,G,ynq,H,I),J)).
can_start(aux(syn(A,B,C,D,E),F),vp(syn(G,H,I,J,n,K,n,L,M,N),O)).
can_start(pred(A,B),pred(C,D)).
can_start(be(A,B),be(C,D)).
can_start(be(syn(A,tnsd,B,C),D),s(syn(tnsd,E,F,ynq,G,H),I)).
can_start(be(syn(A,B,C,D),E),vp(syn(F,G,H,I,n,J,n,K,L,M),N)).
can_start(be(syn(A,B,C,D),E),vp(syn(F,G,H,I,y,J,n,K,L,M),N)).
can_start(pp(A,B),pp(C,D)).
can_start(pp(syn(n,[],[],A,n,[],[],q);B),s(syn(tnsd,C,C,whq,D,D),E)).
can_start(pp(syn(n,A,B,C,n,D,E,n),F),pred(syn(G,H,I,J,K,y,L,M),N)).
can_start(pp(syn(n,[],[],A,n,[],[],r),B),rel(syn(C),D)).
can_start(v(A,B),v(C,D)).
can_start(v(syn(A,B,C,D,np_pp,E),F),vp(syn(G,H,I,J,y,K,L,M,N,O),P)),
can_start(v(syn(A,B,C,D,np_np,E),F),vp(syn(G,H,I,J,y,K,L,M,N,O),P)).
can_start(v(syn(A,B,C,D,trans,E),F),vp(syn(G,H,I,J,y,K,L,M,N,O),P)).
can_start(v(syn(A,B,C,D,intrans,E),F),vp(syn(G,H,I,I,y,J,n,K,L,L),M)).
can_start(adjp(A,B),adjp(C,D)).
can_start(adjp(syn(A,B,C,D),E),pred(syn(F,G,H,I,J,y,K,L),M)).
can_start(adjp(syn([],[],[],[]),A),nbar(syn(B,C,D),E)).
can_start(adjp(syn([],[],[],[]),A),np(syn(n,B,B,y,sing,n,y,C,C,n),D)).
can_start(adjp(syn([],[],[],[]),A),np(syn(n,B,B,n,plur,n,n,C,C,n),D)).
can_start(n(A,B),n(C,D)).
can_start(n(syn(A,B,C,D),E),nbar(syn(F,G,H),I)).
can_start(n(syn(A,B,C,y),D),name(syn(E,F,G),H)).
can_start(n(syn(A,B,C,y),D),np(syn(n,E,E,F,G,n,H,I,I,n),J)).
can_start(nbar(A,B),nbar(C,D)).
can_start(nbar(syn(y,sing,y),A),np(syn(n,B,B,y,sing,n,y,C,C,n),D)).
can_start(nbar(syn(n,plur,n),A),np(syn(n,B,B,n,plur,n,n,C,C,n),D)).
can_start(rel(A,B),rel(C,D)).
can_start(name(A,B),name(C,D)).
can_start(name(syn(A,B,C),D),np(syn(n,E,E,F,G,n,H,I,I,n),J)).
can_start(pro(A,B),pro(C,D)).
can_start(pro(syn(A,B,C,D),E),np(syn(F,G,G,H,I,n,J,K,K,L),M)).
can_start(det(A,B),det(C,D)).
can_start(det(syn(A,B,C,D),E),np(syn(n,F,F,G,H,n,I,J,J,K),L)).
can_start(poss(A,B),poss(C,D)).
can_start(p(A,B),p(C,D)).
can_start(p(syn(A),B),pp(syn(C,D,E,F,G,H,I,J),K)).
can_start(adj(A,B),adj(C,D)).
can_start(adj(syn,A),adjp(syn([],[],[],[]),B)).


rule(s(syn(tnsd,A,B,decl,C,D),E),[np(syn(F,[],[],G,H,n,I,[],[],n),J),vp(syn(F,tnsd,A,B,K,H,n,I
,C,D),L)]).
rule(s(syn(tnsd,A,B,ynq,C,D),E),[aux(syn(F,tnsd,G,H,I),J),np(syn(F,[],[],K,G,n,I,[],[],n),L),v
p(syn(M,H,A,B,N,O,n,n,C,D),P)]).
rule(s(syn(tnsd,A,B,ynq,C,D),E),[be(syn(F,tnsd,G,H),I),np(syn(F,[],[],J,G,n,H,[],[],n),K),pred
(syn(L,M,A,B,N,y,C,D),O)]).
rule(s(syn(tnsd,A,B,ynq,C,D),E),[be(syn(F,tnsd,G,H),I),np(syn(F,[],[],J,G,n,H,[],[],n),K),np(s
yn(L,A,B,M,N,n,O,C,D,n),P)]).
rule(s(syn(tnsd,A,A,whq,B,B),C),[np(syn(D,[],[],E,F,n,G,[],[],q),H),s(syn(tnsd,[np(syn(I,[],[]
,J,F,n,y,[],[],n),K)],[],ynq,[],[g]),L
)]).
rule(s(syn(tnsd,A,A,whq,B,B),C),[pp(syn(n,[],[],D,n,[],[],q),E),s(syn(tnsd,[pp(syn(n,[],[],D,n
,[],[],n),F)],[],ynq,[],[g]),G)]).
rule(s(syn(tnsd,A,A,whq,B,B),C),[np(syn(D,[],[],E,F,n,G,[],[],q),H),vp(syn(D,tnsd,[],[],I,F,n,
G,[],[]),J)]).
rule(vp(syn(A,B,C,D,n,E,n,F,G,H),I),[aux(syn(A,B,E,J,F),K),vp(syn(L,J,C,D,M,N,n,n,G,H),O)]).
rule(vp(syn(A,B,C,D,n,E,n,F,G,H),I),[be(syn(A,B,E,F),J),pred(syn(K,L,C,D,M,y,G,H),N)]).
rule(vp(syn(A,B,C,D,y,E,n,F,G,H),I),[be(syn(A,B,E,F),J),np(syn(K,C,D,L,M,n,N,G,H,n),O)]).
rule(vp(syn(A,B,C,D,y,E,F,G,H,I),J),[v(syn(A,B,K,E,np_pp,G),L),np(syn(M,C,N,O,P,F,Q,H,R,n),S),
pp(syn(n,N,D,K,n,R,I,n),T)]).
rule(vp(syn(A,B,C,D,y,E,F,G,H,I),J),[v(syn(A,B,K,E,np_np,G),L),np(syn(M,[],[],N,O,F,P,[],[],n)
,Q),np(syn(R,C,D,S,T,n,U,H,I,n),V)]).
rule(vp(syn(A,B,C,D,y,E,F,G,H,I),J),[v(syn(A,B,K,E,trans,G),L),np(syn(M,C,D,N,O,F,P,H,I,n),Q)]
).
rule(vp(syn(A,B,C,C,y,D,n,E,F,F),G),[v(syn(A,B,H,D,intrans,E),I)]).
rule(vp(syn(A,B,C,D,y,E,F,G,H,I),J),[vp(syn(A,B,C,K,y,E,F,G,H,L),M),pp(syn(n,K,D,N,n,L,I,n),O)
]).
rule(pred(syn(A,B,C,D,E,F,G,H),I),[vp(syn(A,ing,C,D,y,E,n,F,G,H),J)]).
rule(pred(syn(A,B,C,D,E,F,G,H),I),[vp(syn(A,passive,C,D,y,E,y,F,G,H),J)]).
rule(pred(syn(A,B,C,D,E,y,F,G),H),[adjp(syn(C,D,F,G),I)]).
rule(pred(syn(A,B,C,D,E,y,F,G),H),[pp(syn(n,C,D,I,n,F,G,n),J)]).
rule(nbar(syn(A,B,C),D),[n(syn(E,A,B,C),F)]).
rule(nbar(syn(A,B,C),D),[adjp(syn([],[],[],[]),E),nbar(syn(A,B,C),F)]).
rule(nbar(syn(A,B,C),D),[nbar(syn(A,B,C),E),pred(syn(F,G,[],[],H,y,[],[]),I)]).
rule(nbar(syn(A,B,C),D),[nbar(syn(A,B,C),E),rel(syn(B),F)]).
rule(np(syn(n,A,A,B,C,n,D,E,E,n),F),[name(syn(G,C,D),H)]).
rule(np(syn(A,B,B,C,D,n,E,F,F,G),H),[pro(syn(A,D,E,G),I)]).
rule(np(syn(n,A,A,B,C,n,D,E,E,F),G),[det(syn(B,C,D,F),H),nbar(syn(B,C,D),I)]).
rule(det(syn(A,B,C,q),D),[how,det(syn(A,B,C,n),E)]).
rule(np(syn(n,A,A,y,sing,n,y,B,B,n),C),[nbar(syn(y,sing,y),D)]).
rule(np(syn(n,A,A,n,plur,n,n,B,B,n),C),[nbar(syn(n,plur,n),D)]).
rule(np(syn(n,A,A,B,C,n,D,E,E,F),G),[np(syn(H,[],[],I,J,n,y,[],[],F),K),poss(syn,L),nbar(syn(B
,C,D),M)]).
rule(np(syn(A,[np(syn(B,[],[],C,D,n,y,[],[],n),E)|F],F,G,D,n,y,H,[g|H],n),I),[]).
rule(np(syn(A,B,B,C,D,y,y,E,E,n),F),[]).
rule(pp(syn(n,[pp(syn(n,[],[],A,n,[],[],n),B)|C],C,A,n,D,[g|D],n),E),[]).
rule(pp(syn(A,B,C,D,E,F,G,H),I),[p(syn(D),J),np(syn(K,B,C,L,M,E,N,F,G,H),O)]).
rule(adjp(syn([],[],[],[]),A),[adj(syn,B)]).
rule(rel(syn(A),B),[np(syn(C,[],[],D,A,n,E,[],[],r),F),vp(syn(C,tnsd,[],[],G,A,n,E,[],[]),H)])
.
rule(rel(syn(A),B),[np(syn(C,[],[],D,E,n,y,[],[],r),F),s(syn(tnsd,[np(syn(G,[],[],H,A,n,y,[],[
],n),I)],[],decl,[],[g]),J)]).
rule(rel(syn(A),B),[pp(syn(n,[],[],C,n,[],[],r),D),s(syn(tnsd,[pp(syn(n,[],[],C,n,[],[],n),E)]
,[],decl,[],[g]),F)]).
rule(rel(syn(A),B),[s(syn(tnsd,[np(syn(C,[],[],D,E,n,y,[],[],n),F)],[],decl,[],[g]),G)]).
rule(n(syn(A,B,C,D),E),[n(syn(F,G,H,y),I),n(syn(A,J,C,D),K)]).
rule(name(syn(A,B,C),D),[name(syn(E,F,y),G),name(syn(A,B,C),H)]).
rule(name(syn(A,B,C),D),[name(syn(E,F,y),G),n(syn(A,H,B,C),I)]).
rule(name(syn(A,B,C),D),[n(syn(E,F,G,y),H),name(syn(A,B,C),I)]).
```

```
gap_can_start(np(A,B),np(syn(C,[np(syn(D,[],[],E,F,n,y,[],[],n),G)|H],H,I,F,n,y,J,[g|J],n),K))
.
gap_can_start(np(A,B),np(syn(C,D,D,E,F,y,y,G,G,n),H)).
%gap_can_start(pp(A,B),pp(syn(n,[pp(syn(n,[],[],C,n,[],[],n),D)|E],E,C,n,F,[g|F],n),G)).
gap_can_start(pred(syn(A,B,C,D,E,y,F,G),H),pp(syn(n,[pp(syn(n,[],[],I,n,[],[],n),J)|K],K,I,n,L
,[g|L],n),M)).

cat(wren,name(syn(A,sing,y),sem)).
cat(build,v(syn(A,B,C,D,trans,n),sem(E,F,G,H))).
cat(built,v(syn(A,tnsd,B,C,trans,D),sem(E,F,G,H))).
cat(built,v(syn(I,en,J,K,trans,L),sem(M,N,O,P))).
cat(built,v(syn(Q,passive,R,S,trans,T),sem(U,V,W,X))).
cat(a,det(syn(n,sing,y,n),sem(A,B,C))).
cat(chapel,n(syn(A,B,sing,y),sem)).
cat(college,n(syn(A,B,sing,y),sem)).
cat(in,p(syn(in),sem)).
cat(which,pro(syn(n,A,B,q),sem(C,D))).
cat(which,pro(syn(E,F,B,r),sem(G,H))).
cat(which,det(syn(I,J,K,q),sem(L,M,N))).
cat(did,aux(syn(A,tnsd,B,inf,C),sem(D,E))).
cat(did,v(syn(F,tnsd,G,H,trans,I),sem(J,K,L,M))).
cat(lives,v(syn(A,tnsd,B,sing,intrans,y),sem(C,D,E,F))).
```

8b.5 Adder Source

The adder problem is one of a number of possible queries to a program designed to determine the signal flow through a MOS transistor net. The circuit in this case is a full adder. The method and a description of other signal flow analyses performed by this program are described in Clocksin and Leeser [1986]. This program was adapted for use in this research by the authors of Clocksin and Leeser [1986]. The query for this problem was the following:

problem name	**goal query**
adder	`tdir(adder(A,B,C,D,E),X,Y)`

```
transdir(_,_,_,_,out,in,right).
transdir(_,_,_,_,in,out,left).

tdir(Ckt,P,Dir) :-
        ckt_lookup(Ckt,Clist,Body,Tlist1,_),
        prim_elem(Body,Flat_body,Tlist1,Tlist),
        trydir(Clist,Flat_body,P,Tlist,Dir).

trydir([Name|Ports],Body,Plist,Tlist1,Tlist) :-
                rmv_png(Body,PNG,Rest),
                setdir(Rest,Ports,PNG,[],Plist,[],Intlist,Tlist1,Tlist2),
                flatten(Tlist2,Tlist),
                valid(Intlist,PNG).

rmv_png([],[],[]).
rmv_png([pwr(X)|T],[X|PGlist],Rest) :- rmv_png(T,PGlist,Rest).
rmv_png([gnd(X)|T],[X|PGlist],Rest) :- rmv_png(T,PGlist,Rest).
rmv_png([clock(X)|T],[X|PGlist],Rest) :- rmv_png(T,PGlist,Rest).
rmv_png([H|T],PGlist,[H|Rest]) :- functor(H,trans,_), rmv_png(T,PGlist,Rest).

setdir([],_,_,P,P,I,I,T,T).
setdir([H|T],Ports,PNG,P1,P2,I1,I2,T1,T2) :-
              adj_elems(H,Ports,PNG,P1,P3,I1,I3),
              setdir(T,Ports,PNG,P3,P2,I3,I2,T1,T2).

adj_elems(trans(_,A,B,C,D),Ports,PNG,P1,P2,I1,I2) :-
                  strict_member(B,PNG),
                  transdir(_,A,B,C,out,Y,D),
                  add_elem(Ports,A,out,P1,P3,I1,I3),
                  add_elem(Ports,B,out,P3,P4,I3,I4),
                  add_elem(Ports,C,Y,P4,P2,I4,I2).
adj_elems(trans(_,A,B,C,D),Ports,PNG,P1,P2,I1,I2) :-
                  strict_member(C,PNG),
                  transdir(_,A,B,C,X,out,D),
                  add_elem(Ports,A,out,P1,P3,I1,I3),
                  add_elem(Ports,B,X,P3,P4,I3,I4),
                  add_elem(Ports,C,out,P4,P2,I4,I2).
adj_elems(trans(_,A,B,C,D),Ports,PNG,P1,P2,I1,I2) :-
                  not_strict_member(B,PNG),
                  not_strict_member(C,PNG),
                  transdir(_,A,B,C,X,Y,D),
                  add_elem(Ports,A,out,P1,P3,I1,I3),
                  add_elem(Ports,B,X,P3,P4,I3,I4),
                  add_elem(Ports,C,Y,P4,P2,I4,I2).
add_elem(Ports,A,B,P1,P2,L,L) :- strict_member(A,Ports),
                                 elem(A,B,P1,P2).
add_elem(Ports,A,B,L,L,I1,I2) :- not_strict_member(A,Ports),
                                 elem(A,B,I1,I2).

not_strict_member(_,[]).
not_strict_member(X, [Y|T]) :- X \== Y, not_strict_member(X,T).


elem(A,B,[],[[A,B]]).
elem(A,B,[[H|T1]|T],[[H,B|T1]|T]) :- A==H.
elem(A,B,[[H|T]|T1],[[H|T]|T2]) :- H \== A, elem(A,B,T1,T2).
/*
elem(A,B,[[H|T]],[[H|T], [A,B]]) :- H \== A.
*/

valid([],_).
valid([[H|T1]|T2],PNG) :-  strict_member(H,PNG),
                           all_outs(T1),valid(T2,PNG).
valid([[H|T1]|T2],PNG) :-  legal_list(T1), valid(T2,PNG).

legal_list([in|L]) :- member(out,L).
legal_list([out|L]) :- member(in,L).

all_outs([]).
all_outs([out|T]) :- all_outs(T).

all_ins([]).
all_ins([in|T]) :- all_ins(T).


ckt_lookup(Ckt,Clist,Body,Tlist,Siglist) :-
    module((Ckt :- B),Tlist,Siglist),
    listify(B,Body),
    shape(Ckt,Clist).


prim_comps(Circuit,CL,Tlist1,Tlist2) :-
              ckt_lookup(Circuit,_,Body,Tlist,_),
              prim_elem(Body,CL,Tlist,Tlist3),
              append(Tlist1,Tlist3,Tlist2).

prim_elem([],[],L,L).
prim_elem([pwr(P)|T],[pwr(P)|L],L1,L2) :- prim_elem(T,L,L1,L2).
prim_elem([gnd(G)|T],[gnd(G)|L],L1,L2) :- prim_elem(T,L,L1,L2).
prim_elem([H|T],[H|L],L1,[X|L2]) :-
        primmod(H),
        transdir(H,X),
        prim_elem(T,L,L1,L2).
prim_elem([H|T],CL,L1,L2) :-
        functor(H,Prin,_),
        Prin \== 'pwr', Prin \== 'gnd', Prin \== 'trans',
        prim_comps(H,Subl,L1,L3),
        prim_elem(T,L,L3,L2),
        append(Subl,L,CL).

transdir(trans(_,_,_,_,X),X).

append([],L,L).
append([H|T],Z,[H|L]) :- append(T,Z,L).

listify((A,B),[A|L]) :- listify(B,L).
listify(A,[A]) :- functor(A,_,_).

member(X,[X|_]).
member(X,[Y|T]) :- X\==Y,member(X,T).

strict_member(X,[Y|_]) :- X == Y.
strict_member(X,[Y|T]) :- X\==Y,strict_member(X,T).

flatten(X,Y) :- flatten(X,Y,[]).

flatten([],L,L).
flatten([H|T],L1,L2) :-
    flatten(H,L1,L3),flatten(T,L3,L2).
flatten(X,[X|Z],Z) :- atomic(X), X \== '[]'.
```

```
shape(trans(X,G,S,D,Dir),[trans,X,G,S,D,Dir]).

module(
     (invert(A,B) :- trans(n,A,B,P1,X1),
                     trans(p,A,B,P2,X2),
                     pwr(P2);gnd(P1)),
     [X1,X2],
     [sig(A, [l,l,l,h,h,h,h,h,h,h,h,l,l,l,l,l,l,l,l]),
      sig(B,  [h,h,h,h,h,f,f,l,l,l,l,l,l,l,l,r,r,r,h,h,h,h,h,h])]
).
shape(invert(A,B),[invert,A,B]).

module((buffer(A,C) :- invert(A,B),invert(A,C)),
     [],
     [sig(A, [l,l,l,h,h,h,h,h,h,h,h,l,l,l,l,l,l,l,l]),
      sig(B,  [h,h,h,h,h,f,f,l,l,l,l,l,l,l,l,r,r,r,h,h,h,h,h,h]),
      sig(C,  [l,l,l,l,l,r,r,h,h,h,h,h,h,h,h,f,f,f,l,l,l,l,l,l])]
).
shape(buffer(A,C),[buffer,A,C]).

module((dlatch(D,L,Lbar,Q,Qbar) :- trans(n,L,D,X,X1),
                                   trans(p,Lbar,D,X,X2),
                                   trans(n,Lbar,Q,X,X3),
                                   trans(p,L,Q,X,X4),
                                   invert(X,Qbar),
                                   invert(Qbar,Q)),
                [X1,X2,X3,X4],
                [sig(L,[l,l,h,h,h,h,h,h,h,h,h,h,h,h,h,h,h,h,l,l,l,l,l]),
              sig(Lbar,[h,h,l,l,l,l,l,l,l,l,l,l,l,l,l,l,l,l,h,h,h,h,h]),
                 sig(D,[x,x,x,s,s,s,s,s,s,s,s,s,s,s,s,s,s,s,x,x,x,x]),
                 sig(Q,[x,x,x,x,x,x,x,x,x,x,x,x,x,c,c,c,c,s,s,s,s,s]),
              sig(Qbar,[x,x,x,x,x,x,x,x,x,x,x,x,x,c,c,c,c,s,s,s,s,s])
        ]).
shape(dlatch(A,B,C,D,E),[dlatch,A,B,C,D,E]).


module(
     (tri-state(A,X,En,Enbar) :-
        trans(n,A,N1,P1,X1),
        trans(p,A,N2,P2,X2),
        trans(n,En,N1,X,X3),
        trans(p,Enbar,N2,X,X4),
        pwr(P2),gnd(P1)),
     [X1,X2,X3,X4],
     [sig(En,[l,l,l,l,h,h,h,h,h,h,h,h,h,h,h,h,h,h,h,l,l,l,l,l,l,l]),
   sig(Enbar,[h,h,h,h,l,l,l,l,l,l,l,l,l,l,l,l,l,l,l,h,h,h,h,h,h,h]),
       sig(A,[x,x,x,x,l,l,h,h,h,h,h,h,h,l,l,l,l,l,l,x,x,x,x,x,x,x]),
       sig(X,[z,z,z,z,z,z,l,l,r,r,h,h,h,h,h,f,f,l,l,l,l,z,z,z,z,z])
     ]
).
shape(tri-state(A,B,C,D),[tri-state,A,B,C,D]).

module((adder(A,B,C,SUM,CARRY) :-
        carry_part(A,B,C,NCA,CARRY),
        sum_part(A,B,C,NCA,SUM)),[],_).
shape(adder(A,B,C,SUM,CARRY),[adder,A,B,C,SUM,CARRY]).

module((sum_part(A,B,C,NCA,SUM) :-
        pwr(P),gnd(G),
        trans(p,NCA,T1,P,X1),
        trans(p,C,P,T5,X2),
        trans(p,B,T1,T5,X3),
        trans(p,A,T1,T2,X4),
        trans(p,NCA,T5,T2,X5),
        trans(p,T2,P,SUM,X6),
        trans(n,A,T2,T3,X7),
        trans(n,NCA,T2,T6,X8),
        trans(n,T2,SUM,G,X9),
        trans(n,B,T3,T6,X10),
        trans(n,NCA,T3,G,X11),
        trans(n,C,T6,G,X12)),
   [X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12],_).
shape(sum_part(A,B,C,NCA,CARRY),[sum_part,A,B,C,NCA,CARRY]).
```

```
module((carry_part(A,B,C,NCA,CA) :-
        pwr(P),gnd(G),
        trans(p,A,T1,P,X13),
        trans(p,B,T1,P,X14),
        trans(p,A,T2,P,X15),
        trans(p,C,T1,NCA,X16),
        trans(p,B,T2,NCA,X17),
        trans(p,NCA,P,CA,X18),
        trans(n,C,NCA,T3,X19),
        trans(n,B,NCA,T4,X20),
        trans(n,NCA,CA,G,X21),
        trans(n,A,T3,G,X22),
        trans(n,B,T3,G,X23),
        trans(n,A,T4,G,X24)),
   [X13,X14,X15,X16,X17,X18,X19,X20,X21,X22,X23,X24],_).
shape(carry_part(A,B,C,NCA,CA),[carry_part,A,B,C,NCA,CA]).

primmod(trans(_,_,_,_,_)).
```

8b.6 Matrix Multiplication Source

This matrix multiplication problem is a common Prolog benchmark. The origin of the following Prolog source was an electronic bulletin board. Similar source code can also be found in a paper by Conery and Kibler [1985].

problem name	goal query
mm20	`test20(X).`
mm40	`test40(X).`

```
mm(A,B,C) :-  transpose(B,BT), mmt(A,BT,C).
 /* Product of all rows of A with entire B */
mmt([],_,[]).
mmt([Ai|An],B,[Ci|Cn]) :-  mmc(Ai,B,Ci), mmt(An,B,Cn).
 /* Product of all columns of B with row A */
mmc(_,[],[]).
mmc(A,[Bi|Bn],[Ci|Cn]) :-  ip(A,Bi,Ci), mmc(A,Bn,Cn).
 /* Inner Product of two vectors */
ip([],[],0).
ip([Ai|An],[Bi|Bn],C) :- ip(An,Bn,X), C is X + Ai * Bi.
/* Transpose a matrix */
transpose([[]|_],[]).
transpose(M,[Ci|Cn]) :-  columns(M,Ci,R), transpose(R,Cn).

columns([],[],[]).
columns([[Cii|Cin]|C],[Cii|X],[Cin|Y]) :-  columns(C,X,Y).

/* Examples */

test5(P) :-

mm([[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5],[1,2,3,4,5]],[[5,6,7,8,9],[5,6,7,8,9],[5,6
,7,8,9],[5,6,7,8,9],[5,6,7,8,9]],P).

test10(P) :-

mm([[1,2,3,4,5,1,2,3,4,5],[1,2,3,4,5,1,2,3,4,5],[1,2,3,4,5,1,2,3,4,5],[1,2,3,4,5,1,2,3,4,5],[1
,2,3,4,5,1,2,3,4,5],[1,2,3,4,5,1,2,3,
4,5],[1,2,3,4,5,1,2,3,4,5],[1,2,3,4,5,1,2,3,4,5],[1,2,3,4,5,1,2,3,4,5],[1,2,3,4,5,1,2,3,4,5]],
[[5,6,7,8,9,5,6,7,8,9],[5,6,7
,8,9,5,6,7,8,9],[5,6,7,8,9,5,6,7,8,9],[5,6,7,8,9,5,6,7,8,9],[5,6,7,8,9,5,6,7,8,9],[5,6,7,8,9,5
,6,7,8,9],[5,6,7,8,9,5,6,7,8,9],[5,6,7
,8,9,5,6,7,8,9],[5,6,7,8,9,5,6,7,8,9],[5,6,7,8,9,5,6,7,8,9]],P).

test20(P) :- SIMILAR TO THE CODE ABOVE WITH TWO 20 x 20 MATRICES

test40(P) :- SIMILAR TO THE CODE ABOVE WITH TWO 40 x 40 MATRICES
```

Appendix 9a Portability Notes

9a.1 Technical Note on Signal Catching

It is unfortunate that when a child process dies a clean exit is not always ensured. Often, exiting children remain as *zombie* processes until a higher-up process performs a proper clean up procedure. This clean up includes a method to "reap" its children. If the reap is not performed, each of the children may hold on to a process slot, and there are a limited number of process slots available on a particular host machine.

Exiting children is a common problem for any process which forks off numerous child processes (such as a daemon process) and does not catch the signal SIGCHLD. Very quickly all of the process slots available on that machine are used up, and no more work can be done until the zombie processes (which are each holding on to a process slot) are eliminated. A solution to this problem is demonstrated in *A 4.2BSD Interprocess Communication Primer* [ULTRIX-32 Supplementary Documents: Volume III System Managers 1984]. Here it mentions that the signal SIGCHLD should be caught and a handler invoked which contains the reaper code.

If the signal SIGCHLD is ignored (this is the default action) and child processes are not reaped, then there is a potential of using up all of the available process slots. If the signal SIGCHLD is to be caught, then there is a danger of system calls being interrupted when the signal is received. In particular, the system calls which read and write to slow devices have a high probability of being interrupted if children processes are being created and exiting frequently. For this reason, an extra bit of code needs to be added to most input/output commands such as read, write, accept, recv, and so forth, so that the catching of the SIGCHLD signal does not interfere with the I/O request. The signal handler for catching SIGCHLD and reaping the child process is shown below along with an example of checking for the EINTR error on i/o to slow devices. The EINTR error occurs when a system call is interrupted by a signal which the user has elected to catch. In addition to the SIGCHLD signal, Delphi processes have handlers for the following signals:

SIGHUP	caught to avoid accidentally killing the process
SIGINT	caught to avoid accidentally killing the process
SIGQUIT	caught to avoid accidentally killing the process
SIGALRM	caught to create a timer for certain control strategies
SIGTERM	caught to avoid accidentally killing the process
SIGURG	caught in conjunction with the out of band data byte

The use of any of these signals requires that the EINTR error be checked for during every call to system routines such as read and write.

The following code is the signal handler which catches SIGCHLD and performs a reap of the child process which caused the signal to occur. This ensures that a zombie process is not created when the child exits.

```
void Mourn()
{
  union wait status;

  while (wait3(&status, WNOHANG, 0) > 0)
    ;
}
```

To specify the routine Mourn as the signal handler for SIGCHLD, the following code must be executed before the arrival of any SIGCHLD signal.

```
signal(SIGCHLD, Mourn);
```

An example is given of checking for the EINTR error during a write to a slow device. The EINTR error will occur if signal SIGCHLD arrives during the write call of the following output routine.

```
int bsdwrite(fd, buf, n)
  register int fd, n;
  char *buf;
{
  register int done = 0;
  register int nwritten;

  do {
    if ((nwritten = write(fd, buf+done, n-done)) >= 0)
      done += nwritten;
    else if (errno != EINTR)
    return (nwritten);
  } while (done < n);
  return (done);
}
```

9a.2 Port to Bobcats

A port of most of the Delphi code was made from the µVAXes to a group of HP 9000 Series 350 workstations which are also known as Bobcats. The flavour of UNIX running on these Bobcats is called HP-UX. This operating system contains the Hewlett-Packard implementation of 4.2BSD Interprocess Communication (IPC) facilities and is very similar to the ULTRIX IPC implementation. The most notable changes can be seen in some signal and socket options and with the initiation of signal handlers. The signal caught when a child process dies in particular has been properly implemented in this version. The signal (SIGCLD or SIGCHLD) can be ignored in the HP-UX implementation and no zombie processes will be created. The following Table 9a.1 shows a few of the differences which were discovered during the Delphi port from the µVAXes to the Bobcats.

ULTRIX	HP-UX
bcmp(a,b,c)	memcmp(a,b,c)
bcopy(a,b,c)	memcpy(b,a,c)
bzero(a,b)	memset(a,0,b)
ffs routine	no equivalent
getrusage routine for timings	ftime routine
Signal SIGCHLD	SIGCLD
SIGCHLD should not be ignored else zombie processes are created	SIGCLD can be properly ignored
signal handlers can be assigned with the simplified signal routine. The routine sigvec could also have been used	sigvector must be used to assign handlers and some signal options
sprintf returns a character pointer	sprintf returns the number of bytes

Table 9a.1 Differences in Two UNIX Implementations

Appendix 9b Out of Band Data

9b.1 Using Out of Band Data

The TCP stream socket implementation contains a facility for sending *out of band* data to a process with a socket connection. Special commands must be coded on the receiving side of the connection to initialise the out of band data facilities. A logically independent socket connection is created between the client and the server process after these initialisation instructions have been executed. This special connection is used to send particular signals to the server process in addition to the out of band data. Figure 9b.1 shows the state of the client and server connections after the out of band data facility has been initialised.

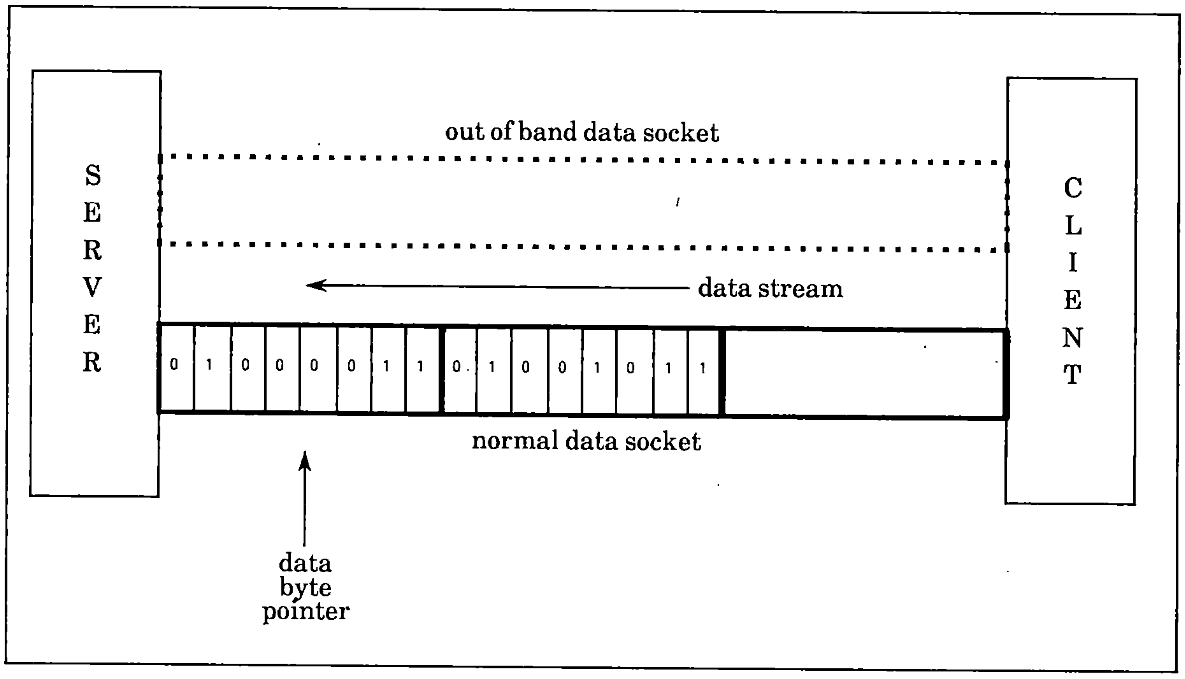


Figure 9b.1 Initialisation of Out of Band Data Facility

The logical communications channel for out of band data is not used until a special `send` command is executed by the client. The flags parameter of the `send` command must be set to `MSG_OOB` to send out of band data to the server process. Either a `send` or a `write` command can be used to send normal data down a socket connection,. When the out of band data is sent three things occur:

(1) The signal `SIGURG` is sent to the server process.

(2) The out of band data byte (this message can only be a single byte) is placed on the logical socket connection.

(3) A marker is placed on the original socket to save the position within the data stream where the out of band data occurred.

Figure 9b.2 shows the socket connections after the client process has sent an out of band data message to the server. Signal SIGURG is sent down the logical connection and can immediately be caught by a handler at the server side. The out of band data, however, may not be immediately available for reading by the server process. All of the data in the stream which occurs before the out of band data marker must first be read.

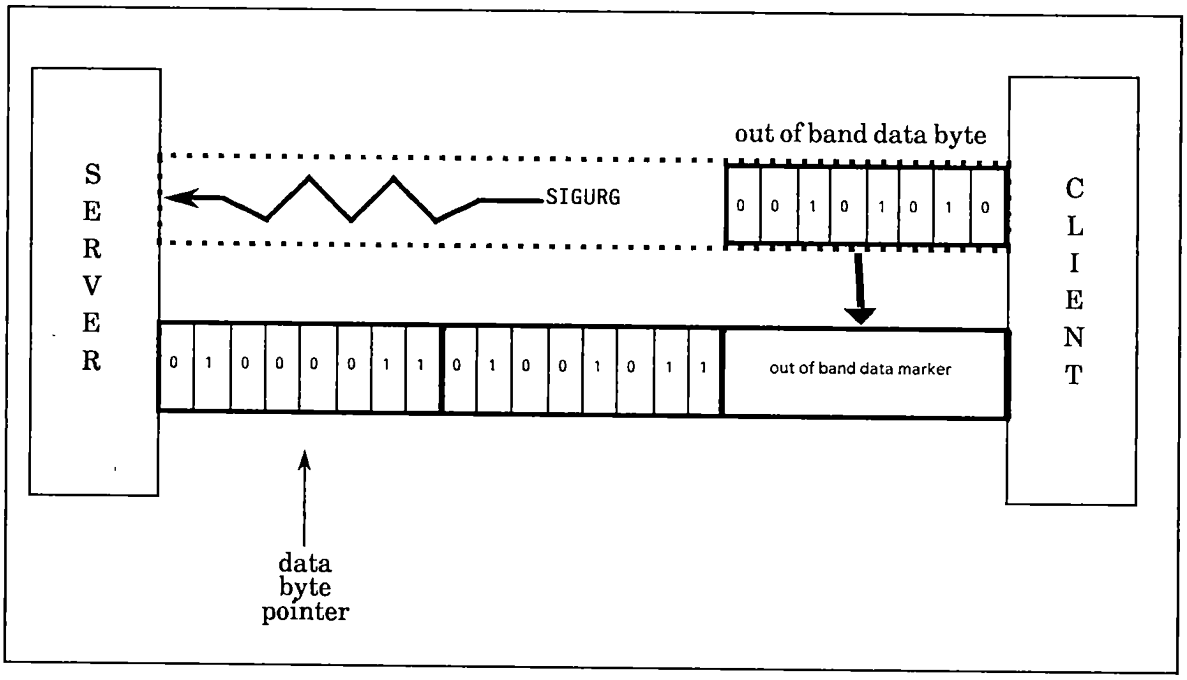


Figure 9b.2 Sending Out of Band Data

Reading data precisely up to the out of band data marker can be automatically performed without the need to handle and check each byte separately. This is done by specifying a large number of bytes to be received in the input routine. When the recv (or read) call tries to read past the out of band data marker, the call will only return the number of bytes up to the marker. The out of band data byte optionally can be read after all of the data in the stream has been read up to the out of band data marker. It is not mandatory to read the out of band data byte. Either the original socket data or the logical socket data (the out of band data byte) can now be read. If the out of band data byte is to be read, a special recv call must be executed with the flags set to MSG_OOB. If a normal recv (or any read call) is executed, then the original socket's data stream will be read. Figure 9b.3 shows the two possibilities for reading data after the data pointer has reached the out of band data marker.

After either socket is read from, the out of band data byte will no longer be available for reading. The byte will disappear from the logical socket and the marker will be taken out of the original socket's data stream. Figure 9b.4 shows the status of the client and server connections after some input routine has been performed.

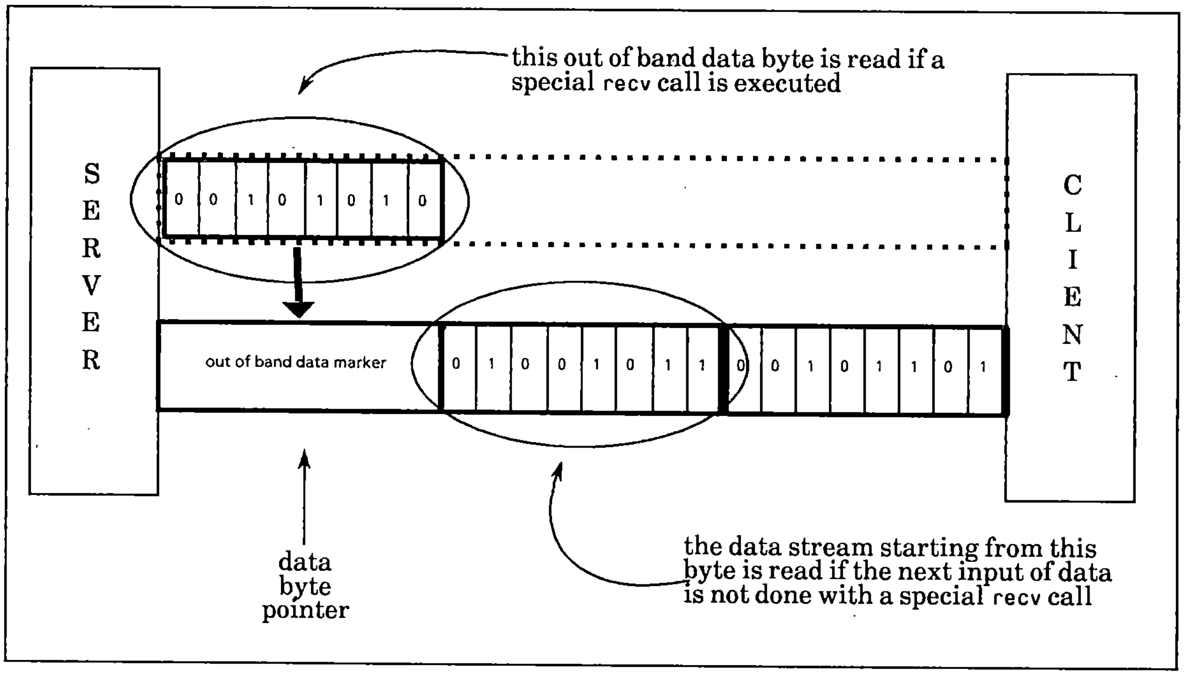

this out of band data byte is read if a special recv call is executed
SERVER
CLIENT
0 0 1 0 1 0 1 0
out of band data marker
0 1 0 0 1 0 1 1
0 0 1 0 1 1 0 1
data byte pointer
the data stream starting from this byte is read if the next input of data is not done with a special recv call

Figure 9b.3 Inputting Data From Either Socket

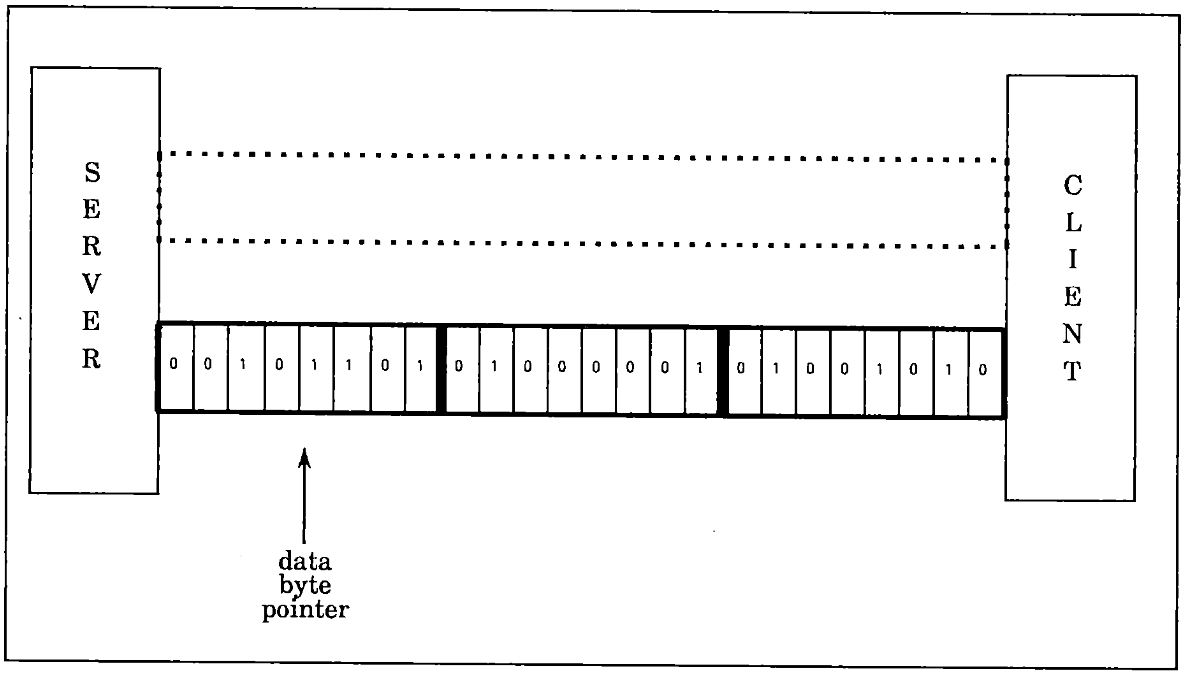


Figure 9b.4 After the Input Routine - Out of Band Data Byte is No Longer Available

Out of band data is often used to signal that the buffer is to be flushed up to the point where the out of band marker has been placed. The signal SIGURG is first received announcing the existence of the out of band data byte. The buffer can then be flushed by performing a read or recv with the number of bytes parameter set equal to the size of the buffer. The data will be read only up to the position of the out of band data marker. The data received after the out of band data marker can now be dealt with as appropriate. Out of band data usage and flushing buffers is documented using HP's implementation of the version 4.2BSD Interprocess Communication facilities in [Using ARPA Services HP 9000 Series 300 1989]. The ULTRIX version of this same facility is documented in *A 4.2BSD Interprocess Communication Primer* contained in [ULTRIX-32 Supplementary Documents: Volume III System Managers 1984]. The documentation available on out of band data is not very thorough. An example program using out of band data on the μVAXes is given in the following two sections of this appendix.

9b.2 Out of Band Data Example - Server Code

The complete code for the out of band data server is shown in Figure 9b.5 and Figure 9b.6.

```
#include <sys/param.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/file.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <netdb.h>
#include <errno.h>

int    ClSock;
/* routine definitions for this file */
void Accepted();
void init__connection();
void ServerLoop();
int sock__int__urg();
typedef int Boolean;
#define      FALSE  0
#define      TRUE   1
Boolean OOBFlag = FALSE;

void Accepted(s)
 int s;
{
  ClSock = s;
} /* end Accepted */

void init__connection__sv()
{
  char service[50];
  int on = 1;
  fprintf(stdout, "\ninit__connection__info: service? ");
  scanf("%s", service);
  ServerLoop(service);
} /* end init__connection__info */

int bsdread(fd, buf, n)
 register int fd, n;
 char *buf;
{
 register int done = 0;
 register int nread;
 do {
  if ((nread = read(fd, buf + done, n-done)) > 0) {
   done + = nread;
  } else if (nread = = 0) {
    return(0);
  } else if (nread < 0) {
    if (errno ! = EINTR) {
      printf("\nbsdread: errno = %d\n", errno);
      if (errno = = ECONNRESET) return(0);
      return (nread);
    }
  }
 } while (done < n);
 return (done);
}
```

```
void ServerLoop(service)
 char *service;
{
 struct sockaddr__in server;
 struct sockaddr__in client;
 struct servent *sp;

 int ls;
 int command, pid, i;
 fprintf(stdout, "<<< Connection pending>>>\n");
 memset((char *)&server, 0, sizeof(server));
 memset((char *)&client, 0, sizeof(client));
 server.sin__family = AF__INET;
 server.sin__addr.s__addr = INADDR__ANY;

 sp = getservbyname(service, "tcp");
 if (sp = = 0) {
  fprintf(stdout, " Service %s/tcp not in file /etc/services\n", service);
  exit(1);
 }
 server.sin__port = sp->s__port;
 ls = socket(AF__INET, SOCK__STREAM, 0);
 if (ls < 0) {
  fprintf(stdout, "\nprolog__same:: socket()\n");
  exit(1);
 }
 if (bind(ls, &server, sizeof(server)) < 0) {
  fprintf(stdout, "\nprolog__same:: bind()\n");
  exit(1);
 }
 if (listen(ls, 5) < 0) {
  fprintf(stdout, "\nprolog__same:: listen()\n");
  exit(1);
 }

 {
 int clientsize = sizeof(client);
 int s = accept(ls, &client, &clientsize);
 if (s < 0) {
  if (errno ! = EINTR) {
    fprintf(stdout, "\nprolog__same:: accept()\n");
    exit(1);
  }
 } else {
  close(ls);
  Accepted(s);
 } /* end if s */
 } /* end block */
} /* end ServerLoop */
```

signal SIGURG will cause an EINTR error. This must be checked for.

Figure 9b.5 Server Code - Standard Routines

Figure 9b.5 shows the standard routines needed to initialise a server process. Figure 9b.6 contains the code specific to using the out of band data facility.

The server is executed on any host machine with the user providing a service name. This name is associated with a port number, and that port will be used to established the socket connection. The server then blocks waiting for data to be sent to it, and outputs the character string which has been received. On receiving out of band data, the server reads the out of band byte and prints out an ASCII representation of the character. It is not mandatory to read the out of band data byte. The code within the handler `sock_int_urg` can be left out if the byte is not needed. The signal `SIGURG` is caught in the handler and the flag `OOBflag` is set. This flag can then be examined at a convenient spot within the server program's code. By setting the flag, an asynchronous event such as a signal can be polled for at a particular known position within the code.

```
int sock_int_urg()
{
    char    oobchar;

    OOBFlag = TRUE;

    if (recv (ClSock, &oobchar, 1, MSG_OOB) <= 0) {
        perror ("sock_int_urg recv()");
        exit (4);
    }
    printf ("\nOOB received  oobchar = %c\n", oobchar);
}


main()
{
    signal (SIGURG, sock_int_urg);
    init_connection_sv ();
    if (fcntl (ClSock, F_SETOWN, getpid ()) < 0) {
        perror ("server main fcntl()");
        exit (4);
    }
    printf ("\nmain: server has been initialised\n");
    for (;;) {
        char buf[100];

        if (bsdread (ClSock, buf, sizeof(buf), 0) < 0) {
            perror("server main recv()");
            exit(4);
        }
        printf ("\nserver main: buf = %s\n", buf);
        if (OOBFlag) {
            OOBFlag = FALSE;
            /* Place in code to perform an
               action after out of band data has
               been received    */
        }
    }
}
```

This code is only necessary if the out of band data character needs to be read

set up a signal handler for signal SIGURG.

set socket to allow receipt of SIGURG.

only to this process id.

receiving 'normal' data

This is where the code woulf be performed after receipt of signal SIGURG.

Figure 9b.6 Server Code - Routines Using Out of Band Data

9b.3 Out of Band Data Example - Client Code

The complete code for the out of band data client is shown in Figure 9b.7 and Figure 9b.8. Figure 9b.7 contains the standard code needed to initialise a client process. Figure 9b.8 contains the code specific to usage of the out of band data facility.

```
#include <sys/param.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/file.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <netdb.h>
#include <errno.h>

/* routine definitions for this file */
void direct__connect();

int    SvSock;

void direct__connect()
{
 struct sockaddr__in client;
 struct sockaddr__in server;
 struct hostent *hp;
 struct servent *sp;

 int s;

 char host[20], service[10];
 memset((char *)&client, 0, sizeof(client));
 memset((char *)&server, 0, sizeof(server));
 server.sin__family = AF__INET;
```

```
 printf("\n    hostname for connect? ");
 scanf("%s", host);
 hp = gethostbyname(host);
 if (hp = = 0) {
  fprintf(stderr, " Host `%s' not in file /etc/hosts\n", host);
  return;
 }
 server.sin__addr.s__addr = ((struct in__addr *)(hp->h__addr))->s__addr;
 printf("\n    service for connect? ");
 scanf("%s", service);
 sp = getservbyname(service, "tcp");
 if (sp = = 0) {
  fprintf(stderr, "Service %s/tcp not in file /etc/services\n", service);
  return;
 }

 server.sin__port = sp->s__port;
 s = socket(AF__INET, SOCK__STREAM, 0);
 if (s < 0) {
  perror("direct__connect__to__prolog: socket()");
  return;
 }
 if (connect(s, &server, sizeof(server)) < 0) {
  perror("direct__connect__to__prolog: connect()");
  close(s);
  return;
 }
 SvSock = s;
 return;
} /* end direct__connect */
```

Figure 9b.7 Client Code - Standard Routines

```
main()
{
    direct_connect ();
    for (;;) {
        char buf[100], oobch;

        printf ("\nany string or just a single '*' for OOB? ");
        scanf ("%s", buf);
        if ((strlen(buf) == 1) && (buf[0] == '*')) {
            oobch = '*';
            if (send (SvSock, &oobch, 1, MSG_OOB) <= 0) {   <------  sending out of band
                perror ("client main send() OOB");                   data byte '*' and
                exit (4);                                            signal SIGURG.
            }
        } else {
            if (write (SvSock, buf, sizeof (buf)) <= 0) {   <----  sending 'normal' data
                perror ("client main write()");
                exit (4);
            }
        }
    }           /* end forever */
}
```

Figure 9b.8 Client Code - Main Routine

The client program can be executed on any host machine. Two initialisation parameters must be supplied by the user to establish a connection to the server process:

- Host name of the machine where the server code has been started.
- Name of the service or port which the server code is waiting on for a connection.

After these two parameters are given, the connection is established and the client loop begins execution. The user is prompted continually for any string of data. If the special string "*" followed by a carriage return is entered, then out of band data will be sent to the server process. Any other string will be sent to the server without special treatment.

Appendix 10a Tree Analysis

10a.1 Two Queens Problem

```
                DEPTHANALYSIS

Number of answers = 0
Number of branches = 10, Maximum branch length = 6, Minimum branch length = 1
Number of branches of length 1 = 1
Number of branches of length 3 = 2
Number of branches of length 4 = 2
Number of branches of length 5 = 3
Number of branches of length 6 = 2
Average branch length = 4
Most common length = 5

                BREADTHANALYSIS

Maximum branching factor = 2
Total number of nodes = 19: internal nodes = 9
     nondeterministic = 9, deterministic = 0
Number of leaves = 10: failed leaves = 10, answers = 0
Maximum width (number of nodes) on a level = 4 on level 3
```

```
   2______0
  F_2_____1
  2_2_____2
F_2_2_F___3
F_2_F_2___4
F_F_F_2___5
  F_F_____6
```

10a.2 Three Queens Problem

```
                DEPTHANALYSIS

Number of answers = 0
Number of branches = 29, Maximum branch length = 12, Minimum branch length = 1

Number of branches of length 1 = 1      Number of branches of length 8 = 4
Number of branches of length 3 = 1      Number of branches of length 9 = 1
Number of branches of length 4 = 3      Number of branches of length 10 = 4
Number of branches of length 5 = 3      Number of branches of length 11 = 3
Number of branches of length 6 = 2      Number of branches of length 12 = 2
Number of branches of length 7 = 5

Average branch length = 7
Most common length = 7

                BREADTHANALYSIS

Maximum branching factor = 2
Total number of nodes = 57: internal nodes = 28
     nondeterministic = 28, deterministic = 0
Number of leaves = 29: failed leaves = 29, answers = 0
Maximum width (number of nodes) on a level = 8 on level 7
```

```
       2__________0
      F_2_________1
      2_2_________2
    F_2_2_2_______3
  F_2_F_2_2_F_____4
  F_2_F_2_F_2_____5
  2_F_F_2_2_2_____6
2_F_F_F_2_F_F_2___7
  F_2_F_2_F_F_____8
    2_2_F_2_______9
  F_F_F_2_F_2_____10
    F_F_2_F_______11
      F_F_________12
```

10a.3 Four Queens Problem

```
              DEPTHANALYSIS

Number of answers = 2
Number of branches = 108, Maximum branch length = 21, Minimum branch length = 1

Number of branches of lengths 1 = 1      Number of branches of lengths 12 = 9
Number of branches of lengths 3 = 1      Number of branches of lengths 13 = 11
Number of branches of lengths 4 = 2      Number of branches of lengths 14 = 9
Number of branches of lengths 5 = 4      Number of branches of lengths 15 = 6
Number of branches of lengths 6 = 2      Number of branches of lengths 16 = 2
Number of branches of lengths 7 = 5      Number of branches of lengths 17 = 6
Number of branches of lengths 8 = 6      Number of branches of lengths 18 = 6
Number of branches of lengths 9 = 8      Number of branches of lengths 19 = 10
Number of branches of lengths 10 = 9     Number of branches of lengths 20 = 2
Number of branches of lengths 11 = 5     Number of branches of lengths 21 = 4

Average branch length = 12
Most common length = 13
Number of answers at depth 21 = 2

              BREADTHANALYSIS

Maximum branching factor = 2
Total number of nodes = 215: internal nodes = 107
     nondeterministic = 107, deterministic = 0
Number of leaves = 108: failed leaves = 106, answers = 2
Maximum width (number of nodes) on a level = 18 on level 12
```

```
                  2___________________0
                F_2___________________1
                2_2___________________2
              F_2_2_2_________________3
            F_2_F_2_2_2_______________4
          F_2_F_2_F_2_2_F_____________5
          2_2_F_2_2_2_F_2_____________6
      2_F_2_F_F_2_2_F_F_2_2_2_________7
    F_2_2_F_2_F_F_2_F_2_2_F_2_2_______8
  2_2_F_2_2_F_F_2_F_F_F_2_2_F_F_2_____9
  F_F_F_2_2_2_F_2_F_2_F_2_F_2_F_F_____10
    F_2_F_F_2_2_2_2_2_2_F_2_F_2_______11
F_F_F_2_F_2_F_2_2_2_F_F_2_F_2_2_F_2___12
2_F_F_F_2_F_F_F_F_2_F_2_F_2_2_F_F_2___13
    F_2_F_2_F_F_2_F_2_F_F_F_2_F_______14
        F_2_F_2_F_2_F_2_F_F___________15
          F_2_F_2_2_2_2_2_____________16
      F_2_2_2_F_F_2_2_F_F_F_2_________17
      2_F_F_2_2_F_F_2_F_2_F_2_________18
      F_F_F_2_F_F_F_2_F_F_F_F_________19
              2_F_2_F_________________20
              A_F_A_F_________________21
```

10a.4 Eight Queens Problem

```
                DEPTHANALYSIS

Number of answers = 92
Number of branches = 44396, Maximum branch length = 78, Minimum branch length = 1

Number of branches of lengths 1 = 1       Number of branches of lengths 41 = 1001
Number of branches of lengths 3 = 1       Number of branches of lengths 42 = 1055
Number of branches of lengths 4 = 2       Number of branches of lengths 43 = 1172
Number of branches of lengths 5 = 3       Number of branches of lengths 44 = 1294
Number of branches of lengths 6 = 2       Number of branches of lengths 45 = 1379
Number of branches of lengths 7 = 5       Number of branches of lengths 46 = 1396
Number of branches of lengths 8 = 6       Number of branches of lengths 47 = 1309
Number of branches of lengths 9 = 11      Number of branches of lengths 48 = 1243
Number of branches of lengths 10 = 15     Number of branches of lengths 49 = 1155
Number of branches of lengths 11 = 19     Number of branches of lengths 50 = 1080
Number of branches of lengths 12 = 26     Number of branches of lengths 51 = 1017
Number of branches of lengths 13 = 32     Number of branches of lengths 52 = 1144
Number of branches of lengths 14 = 45     Number of branches of lengths 53 = 1178
Number of branches of lengths 15 = 50     Number of branches of lengths 54 = 1280
Number of branches of lengths 16 = 65     Number of branches of lengths 55 = 1300
Number of branches of lengths 17 = 87     Number of branches of lengths 56 = 1310
Number of branches of lengths 18 = 95     Number of branches of lengths 57 = 1373
Number of branches of lengths 19 = 118    Number of branches of lengths 58 = 1215
Number of branches of lengths 20 = 137    Number of branches of lengths 59 = 1097
Number of branches of lengths 21 = 177    Number of branches of lengths 60 = 873
Number of branches of lengths 22 = 171    Number of branches of lengths 61 = 770
Number of branches of lengths 23 = 250    Number of branches of lengths 62 = 667
Number of branches of lengths 24 = 234    Number of branches of lengths 63 = 601
Number of branches of lengths 25 = 327    Number of branches of lengths 64 = 506
Number of branches of lengths 26 = 312    Number of branches of lengths 65 = 741
Number of branches of lengths 27 = 416    Number of branches of lengths 66 = 712
Number of branches of lengths 28 = 390    Number of branches of lengths 67 = 895
Number of branches of lengths 29 = 485    Number of branches of lengths 68 = 746
Number of branches of lengths 30 = 463    Number of branches of lengths 69 = 697
Number of branches of lengths 31 = 577    Number of branches of lengths 70 = 565
Number of branches of lengths 32 = 569    Number of branches of lengths 71 = 491
Number of branches of lengths 33 = 719    Number of branches of lengths 72 = 404
Number of branches of lengths 34 = 755    Number of branches of lengths 73 = 401
Number of branches of lengths 35 = 864    Number of branches of lengths 74 = 148
Number of branches of lengths 36 = 872    Number of branches of lengths 75 = 78
Number of branches of lengths 37 = 916    Number of branches of lengths 76 = 33
Number of branches of lengths 38 = 911    Number of branches of lengths 77 = 18
Number of branches of lengths 39 = 948    Number of branches of lengths 78 = 8
Number of branches of lengths 40 = 968

Average branch length = 48
Most common length = 75
Number of answers at depth 73 = 92

                BREADTHANALYSIS

Maximum branching factor = 2
Total number of nodes = 88791: internal nodes = 44395
     nondeterministic = 44395, deterministic = 0
Number of leaves = 44396: failed leaves = 44304, answers = 92
Maximum width (number of nodes) on a level = 2700 on level 45
```

10a.5 Parser-2 Problem

```
                DEPTHANALYSIS

Number of answers = 5
Number of branches = 30911, Maximum branch length = 81, Minimum branch length = 1

Number of branches of length 1 = 2      Number of branches of length 42 = 289
Number of branches of length 2 = 3      Number of branches of length 43 = 108
Number of branches of length 3 = 40     Number of branches of length 44 = 1147
Number of branches of length 4 = 8      Number of branches of length 45 = 243
Number of branches of length 5 = 2      Number of branches of length 46 = 423
Number of branches of length 6 = 1      Number of branches of length 47 = 456
Number of branches of length 7 = 41     Number of branches of length 48 = 1038
Number of branches of length 8 = 5      Number of branches of length 49 = 1253
Number of branches of length 9 = 3      Number of branches of length 50 = 326
Number of branches of length 10 = 2     Number of branches of length 51 = 555
Number of branches of length 11 = 6     Number of branches of length 52 = 835
Number of branches of length 12 = 252   Number of branches of length 53 = 1160
Number of branches of length 13 = 15    Number of branches of length 54 = 498
Number of branches of length 14 = 3     Number of branches of length 55 = 329
Number of branches of length 15 = 3     Number of branches of length 56 = 903
Number of branches of length 16 = 126   Number of branches of length 57 = 579
Number of branches of length 17 = 282   Number of branches of length 58 = 1404
Number of branches of length 18 = 11    Number of branches of length 59 = 390
Number of branches of length 19 = 257   Number of branches of length 60 = 1176
Number of branches of length 20 = 133   Number of branches of length 61 = 768
Number of branches of length 21 = 78    Number of branches of length 62 = 749
Number of branches of length 22 = 22    Number of branches of length 63 = 1151
Number of branches of length 23 = 255   Number of branches of length 64 = 679
Number of branches of length 24 = 282   Number of branches of length 65 = 704
Number of branches of length 25 = 45    Number of branches of length 66 = 429
Number of branches of length 26 = 138   Number of branches of length 67 = 1138
Number of branches of length 27 = 29    Number of branches of length 68 = 297
Number of branches of length 28 = 509   Number of branches of length 69 = 1060
Number of branches of length 29 = 39    Number of branches of length 70 = 519
Number of branches of length 30 = 188   Number of branches of length 71 = 329
Number of branches of length 31 = 150   Number of branches of length 72 = 959
Number of branches of length 32 = 387   Number of branches of length 73 = 669
Number of branches of length 33 = 555   Number of branches of length 74 = 336
Number of branches of length 34 = 18    Number of branches of length 75 = 328
Number of branches of length 35 = 135   Number of branches of length 76 = 558
Number of branches of length 36 = 372   Number of branches of length 77 = 158
Number of branches of length 37 = 333   Number of branches of length 78 = 520
Number of branches of length 38 = 162   Number of branches of length 79 = 36
Number of branches of length 39 = 273   Number of branches of length 80 = 118
Number of branches of length 40 = 626   Number of branches of length 81 = 258
Number of branches of length 41 = 245

Average branch length = 54
Most common length = 58
Number of answers at depth 74 = 1
Number of answers at depth 77 = 2
Number of answers at depth 80 = 2

                BREADTHANALYSIS

Maximum branching factor = 43
Total number of nodes = 33592: internal nodes = 2681
     nondeterministic = 2681, deterministic = 0
Number of leaves = 30911: failed leaves = 30906, answers = 5
Maximum width (number of nodes) on a level = 1479 on level 58
```

10a.6 Parser-4 Problem

```
                DEPTHANALYSIS

Number of answers = 42
Number of branches = 395456, Maximum branch length = 131, Minimum branch length = 1

Number of branches of length 1 = 2       Number of branches of length 67 = 1687
Number of branches of length 2 = 3       Number of branches of length 68 = 1824
Number of branches of length 3 = 40      Number of branches of length 69 = 3484
Number of branches of length 4 = 8       Number of branches of length 70 = 1506
Number of branches of length 5 = 2       Number of branches of length 71 = 1154
Number of branches of length 6 = 1       Number of branches of length 72 = 3683
Number of branches of length 7 = 41      Number of branches of length 73 = 2703
Number of branches of length 8 = 5       Number of branches of length 74 = 4206
Number of branches of length 9 = 3       Number of branches of length 75 = 1441
Number of branches of length 10 = 2      Number of branches of length 76 = 5811
Number of branches of length 11 = 6      Number of branches of length 77 = 2285
Number of branches of length 12 = 252    Number of branches of length 78 = 4189
Number of branches of length 13 = 15     Number of branches of length 79 = 2856
Number of branches of length 14 = 3      Number of branches of length 80 = 4825
Number of branches of length 15 = 3      Number of branches of length 81 = 8118
Number of branches of length 16 = 126    Number of branches of length 82 = 2565
Number of branches of length 17 = 282    Number of branches of length 83 = 5562
Number of branches of length 18 = 11     Number of branches of length 84 = 4358
Number of branches of length 19 = 257    Number of branches of length 85 = 9677
Number of branches of length 20 = 133    Number of branches of length 86 = 4204
Number of branches of length 21 = 78     Number of branches of length 87 = 3991
Number of branches of length 22 = 22     Number of branches of length 88 = 6902
Number of branches of length 23 = 255    Number of branches of length 89 = 7736
Number of branches of length 24 = 282    Number of branches of length 90 = 12445
Number of branches of length 25 = 45     Number of branches of length 91 = 3810
Number of branches of length 26 = 138    Number of branches of length 92 = 9308
Number of branches of length 27 = 29     Number of branches of length 93 = 6164
Number of branches of length 28 = 509    Number of branches of length 94 = 11236
Number of branches of length 29 = 39     Number of branches of length 95 = 7926
Number of branches of length 30 = 188    Number of branches of length 96 = 5159
Number of branches of length 31 = 150    Number of branches of length 97 = 9246
Number of branches of length 32 = 387    Number of branches of length 98 = 7728
Number of branches of length 33 = 555    Number of branches of length 99 = 14524
Number of branches of length 34 = 18     Number of branches of length 100 = 4810
Number of branches of length 35 = 135    Number of branches of length 101 = 11598
Number of branches of length 36 = 372    Number of branches of length 102 = 6695
Number of branches of length 37 = 333    Number of branches of length 103 = 9241
Number of branches of length 38 = 162    Number of branches of length 104 = 9573
Number of branches of length 39 = 273    Number of branches of length 105 = 6742
Number of branches of length 40 = 626    Number of branches of length 106 = 8021
Number of branches of length 41 = 245    Number of branches of length 107 = 4727
Number of branches of length 42 = 289    Number of branches of length 108 = 13909
Number of branches of length 43 = 108    Number of branches of length 109 = 3882
Number of branches of length 44 = 1147   Number of branches of length 110 = 10783
Number of branches of length 45 = 243    Number of branches of length 111 = 6534
Number of branches of length 46 = 423    Number of branches of length 112 = 5344
Number of branches of length 47 = 456    Number of branches of length 113 = 9649
Number of branches of length 48 = 1038   Number of branches of length 114 = 7304
Number of branches of length 49 = 1253   Number of branches of length 115 = 5283
Number of branches of length 50 = 326    Number of branches of length 116 = 3164
Number of branches of length 51 = 555    Number of branches of length 117 = 9725
Number of branches of length 52 = 835    Number of branches of length 118 = 2353
Number of branches of length 53 = 1160   Number of branches of length 119 = 7972
Number of branches of length 54 = 486    Number of branches of length 120 = 4455
Number of branches of length 55 = 341   Number of branches of length 121 = 2525
Number of branches of length 56 = 1407  Number of branches of length 122 = 6975
Number of branches of length 57 = 627   Number of branches of length 123 = 4953
Number of branches of length 58 = 1374  Number of branches of length 124 = 2464
Number of branches of length 59 = 420   Number of branches of length 125 = 2296
Number of branches of length 60 = 2424  Number of branches of length 126 = 3906
Number of branches of length 61 = 876   Number of branches of length 127 = 1106
Number of branches of length 62 = 1265  Number of branches of length 128 = 3640
Number of branches of length 63 = 1229  Number of branches of length 129 = 252
Number of branches of length 64 = 2224  Number of branches of length 130 = 826
Number of branches of length 65 = 3185  Number of branches of length 131 = 1806
Number of branches of length 66 = 903

Average branch length = 95
Most common length = 99
Number of answers at depth 118 = 1
Number of answers at depth 121 = 4
Number of answers at depth 124 = 9
Number of answers at depth 127 = 14
Number of answers at depth 130 = 14
```

10a.7 Larger Problems

The 2-queens, 3-queens, and 4-queens problems are small enough that an ASCII rendering can be displayed on a single A4 sheet of paper. The 8-queens ASCII tree takes up a few sheets of A4, and has not been shown. The parser problem trees (parser-2, parser-3, and parser-4) are also very large, with the parser-4 problem large enough that the breadth-first analysis was not completed. This is because the sorting process on the depth-first tree takes a very long time to complete. Both the adder and the pentominoes problems have trees much larger than any of the previously described programs. The data from these two large problems has not been shown for reasons either of space within this document, or available disk space to perform the analysis and sorting algorithms. Another program was written which performs a subset of the full analysis and can be used with problems of any size. This program finds two important details of any Prolog search space: the number of solutions, and the maximum depth of the tree. This program was run using most of the Prolog problems described in this dissertation. The results are shown in Table 10a.1.

problem	number of solutions	maximum depth of the OR-only tree
adder	8	1227
mm20	1	36
mm40	1	1681
ortest `c(10)`	10	5002
ortest `c(50)`	50	5002
ortest `c(100)`	100	5002
parser-2	5	81
parser-3	14	106
parser-4	42	131
pentominoes	16	323
2-queens	0	6
3-queens	0	12
4-queens	2	21
8-queens	92	78
9-queens	352	96
10-queens	724	118

Table 10a.1 Vital Statistics for All Problems

Bibliography

Ali, K.A.M., *Pool machine: a multiprocessor architecture for OR-parallel execution of logic programs.* The Royal Institute of Technology, Stockholm, TRITA-CS-8603, 1985.

Ali, K.A.M., OR-parallel execution of Prolog on a multi-sequential machine. *International Journal of Parallel Programming,* 15(3), pp. 189-214, 1987.

Ali, K.A.M., Fahlen L., and Karlsson R., *The BC-Machine: A multiprocessor architecture for fast OR-parallel execution of logic programs.* Swedish Institute of Computer Science, Reproduced from a set of overhead slides, 1986.

Ali, K.A.M., and Wong, M., *An investigation of an OR parallel execution model for Horn clause programs.* Swedish Institute of Computer Science, Stockholm, 1988.

Alshawi, H., and Moran, D.B., The Delphi model and some preliminary experiments. *Proc. of the Fifth international Conference and Symposium,* MIT Press, pp. 1578-1589, 1988.

ARPA/Berkeley Services Reference Pages HP 9000 Series 300. Manual Part Number: 50952-90031, Hewlett-Packard Company, Fort Collins, CO, USA, 1987.

Bic, L., Execution of logic programs on a dataflow architecture. *Proceedings of the 11th Annual International Symposium on Computer Architecture,* ACM, Ann Arbor, pp. 290-296, 1984.

Borgwardt, P., Parallel Prolog using stack segments on shared-memory multiprocessors. *Proceedings 1984 International Symposium Logic Programming,* pp. 2-11, 1984.

Burns, A., *Programming in Occam 2.* Addison-Wesley Publishing Company, 1988.

Butler, R., Lusk, E.L., Olsan, R., and Overbeek, R.A., *ANLWAM: A parallel implementation of the Warren Abstract Machine,* Internal Report, Argonne National Laboratory, 1986.

Carlton, M. and Van Roy, P., A distributed Prolog system with AND-parallelism. *IEEE Software,* pp. 43-51, January 1988.

Chang, J-H., Despain, A.M., and DeGroot, D., AND-parallelism of logic programs based on static data dependency analysis. Digest of papers of *COMPCON Spring '85*, pp. 218-225, 1985.

Ciepielewski, A., and Haridi, S., A formal model for OR-parallel execution of logic programs. *Information Processing 83*, pp. 299-305, 1983.

Clark, K.L., and Gregory, S., A relational language for parallel programming. In *Proceedings of the. 1981 Conference on Functional Programming Languages and Computer Architectures*, ACM, pp. 171-178, 1981.

Clark, K.L., and Gregory, S., PARLOG: parallel programming in logic. *ACM Trans. on Prog. Lang. and Systems*, 8(1), pp. 1-49, 1986.

Clark, K.L., McCabe, F.G. and Gregory, S., IC-Prolog language features, In Clark and Tärnlund, editors, *Logic Programming*, Academic Press, 1982.

Clark, K.L., and Tärnlund, S. A., editors, *Logic Programming*, Academic Press, 1982.

Clocksin, W.F., Principles of the DelPhi parallel inference machine. *Computer Journal* 30(5), pp. 386-392, 1987.

Clocksin, W.F., and Alshawi, H., A method for efficiently executing horn clause programs using multiple processors, *New Generation Computing*, 5, pp. 361-376, 1988.

Clocksin, W.F., and Leeser, M.E., Automatic determination of signal flow through MOS transistor networks. *Integration*, 4, pp. 53-63, 1986.

Clocksin, W.F., and Mellish, C.S., *Programming in Prolog*, Springer-Verlag, 1981.

Codish, M. and Shapiro, E.H., Compiling OR-parallelism into AND-parallelism. *Third International Conference on Logic Programming*, London, pp. 283-297, 1986.

Conery, J.S., *The AND/OR process model for parallel execution of logic programs*. PhD thesis, University of California, Irvine, 1983.

Conery, J.S., *Parallel execution of logic programs*. Kluwer Academic Publishers, Boston, 1987.

Conery, J.S., and Kibler, D.F., Parallel interpretation of logic programs. *Proceedings of the 1981 Conference on Functional Programming and Computer Architecture*, pp. 163-170, 1981.

Conery, J.S., and Kibler, D.F., AND parallelism and nondeterminism in logic programs. *New Generation Computing*, 3, pp. 43-70, 1985.

Crammond, J., A comparative study of unification algorithms for OR-parallel execution of logic languages. *IEEE Transactions on Computers*, C-34, pp. 911-917, 1985.

DeGroot, D., Restricted AND-parallelism. In *2nd International Conference on Fifth Generation Computer Systems*, ICOT, pp. 471-478, 1984.

Dijkstra, E.W., *A Discipline of Programming*. Englewood Cliffs, NJ, Prentice-Hall, 1976.

Dwork, C., Kanellakis, P.C., and Mitchell, J.C., On the sequential nature of unification. *Journal of Logic Programming*, 1, pp. 35-50, 1984.

Finkel, R., and Manber, U., DIB - A distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2), pp. 235-256, 1987.

2nd International Conference on Fifth Generation Computer Systems, Nov., Tokyo, Japan, Aiso, H., editor, Elsevier North Holland, Amsterdam, 1984.

Fuchi, K., and Furukawa, K., The role of logic programming in the Fifth Generation Computer Project. *Third International Conference on Logic Programming*, pp. 1-24, 1983.

Furukawa, K., Nitta, K., and Matsumoto, Y., Prolog interpreter based on concurrent programming. In *1st International Logic Programming Conference*, pp. 38-44, 1982.

Gehani, N., and McGettrick, A.D., editors, *Concurrent programming*. Addison-Wesley, 1988.

Goto, A., Tanaka, H., and Moto-Oka, T., Highly parallel inference engine PIE: Goal rewriting model and machine architecture. New Generation Computing, 2, pp. 37-58, 1984.

Gregory, S., Foster, I.T., Burt, A.D., Ringwood, G.A., An abstract machine for the implementation of PARLOG on uniprocessors. *New Generation Computing*, 6, pp. 389-420, 1989.

Guest, S., *Delphi Checkpointing.* Diploma Project in Computer Science, University of Cambridge, England, 1989.

Halim, Z, A data-driven machine for OR-parallel evaluation of logic programs. *New Generation Computing,* 4, pp. 5-33, 1986.

Hardi, S., and Ciepielewski, A., An OR-parallel Token Machine. *Proceedings Logic Programming Workshop'83,* pp. 537-552, 1983.

Hermenegildo, M.V., An abstract machine for restricted AND-parallel execution of logic programs. *Proc. 3rd International Logic Programming Conference,* In *Lecture Notes for Computer Science,* Vol. 225, pp. 25-39, Springer-Verlag, 1986.

Hermenegildo, M.V., and Nasr R.I., Efficient management of backtracking in AND-parallelism. *Proc. 3rd International Logic Programming Conference,* In *Lecture Notes for Computer Science,* Vol. 225, pp. 40-54, Springer-Verlag, 1986.

Hoare, C.A.R., Communicating sequential processes. *Communications of the ACM,* 17(10), pp. 549-557, 1978.

Horowitz, E., and Zorat, A., Divide-and-conquer for parallel processing. *IEEE Transactions on Computers* C-32, pp. 582-585, 1983.

Hwang, K., and Briggs, F.A., *Computer architecture and parallel processing.* McGraw-Hill, 1984.

1st International Logic Programming Conference, Faculte des Sciences de Luminy, Marseille, France, September 14-17, 1982.

Kahn, G., and MacQueen, D.B., Coroutines and networks of parallel processes. *Information Processing 77; Proceedings of the IFIP Congress 77,* B. Gilchrist (Ed.), Amsterdam, pp. 993-998, 1977.

Kasif, S., Kohli, M., and Minker, J., Prism - a parallel inference system for problem solving. In *2nd International Workshop on Logic Programming,* pp. 123-152, 1983.

Kasif, S., and Minker, J., The Intelligent Channel: a scheme for result sharing in logic programs. *9th International Joint Conference on Artificial Intelligence,* Los Angeles, USA, pp. 29-31, 1984.

Kennaway, J.R., and Sleep, M.R., Novel architectures for declarative languages. *Software and Microsystems* 2(3), pp. 59-70, 1983.

Korf, R.E., Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27, pp. 97-109, 1985.

Kowalski, R., *Logic for Problem Solving.* North Holland, 1979.

Leffler, S.J., McKusick, M.K., Karels, M.J., and Quarterman, J.S. *The Design and Implementation of the 4.3BSD Operating System.* Addison-Wesley Publishing Company, 1989.

Levy, J., Shared memory execution of committed-choice languages. *Third International Conference on Logic Programming,* London, pp. 298-312, 1986.

Lin, Y., Kumar, V., and Leung, C., An intelligent backtracking algorithm for parallel execution of logic programs. *Proc. 3rd International Logic Programming Conference,* In *Lecture Notes for Computer Science,* Vol. 225, pp. 55-68, Springer-Verlag, 1986.

Lindstrom, G., OR-parallelism on applicative architectures. *Proceedings 2nd International Logic Programming Conference,* pp. 159-170, 1984.

Lusk, E., Overbeek, R., *et al., Portable programs for parallel processors.* Holt Rinehart and Winston, 1987.

Milner, R., A Calculus of Communicating Systems. *Lecture Notes in Computer Science,* 92, Springer-Verlag, New York, 1980.

Monteiro, L., A horn clause like logic for specifying concurrency. *Proc. of the Second International Logic Programming Conference,* pp. 1-8, 1982.

Mullender, S.J., editor, *The Amoeba distributed operating system: selected papers 1984-1987.* Stichting Mathematisch Centrum, Amsterdam, 1987.

Okumura, A., and Matsumoto, Y., Parallel programming with layered streams. *Proc. 1987 International Symposium on Logic Programming,* pp. 224-231, 1987.

Onai, R., Moritoshi, A., Shimizu, H., Masuda, K., and Matsumoto, A., Architecture of a reduction-based parallel inference machine: PIM-R. *New Generation Computing,* 3, pp. 197-228, 1985.

Overbeek, R.A., *et al., Prolog on multiprocessors.* Internal Report, Argonne National Laboratory, 1985.

Papadopoulos, G.A., *Parallel execution of logic programs: a survey.* Internal Report SYS-C87-08, School of Information Systems, University of East Anglia, 1987.

Parke, H.W., and Worwell, D.E.W., *The Delphic Oracle Volume I The History.* Oxford, 1956.

Pereira, F., editor, *C-Prolog User's Manual Version 1.4.* SRI International, Menlo Park, California, 1984.

Pereira, L.M., Monteiro, L., Cunha, J. and Aparicio, J.N., Delta Prolog: a distributed backtracking extension with events. *Proc. 3rd International Logic Programming Conference,* In *Lecture Notes for Computer Science,* Vol. 225, pp. 710-717, Springer-Verlag, 1986.

Porto, A., Epilog: a language for extended programming in logic. In *1st International Logic Programming Conference*, pp. 31-37, 1982.

Saraswat, V.A., The concurrent logic programming language CP: definition and operational semantics. *Conf. Record 14th Annual ACM Symposium on Principles of Programming Languages*, pp. 49-63, 1987.

Shapiro, E.Y., *A subset of Concurrent Prolog and its interpreter.* Technical Report TR-003, ICOT, Tokyo, 1983.

Shapiro, E.Y., Concurrent Prolog: a progress report. *IEEE Computer,* 19(8), pp. 44-58, 1986.

Shapiro, E.Y., An OR-parallel execution algorithm for Prolog and its FCP implementation. *Proc. of the 4th International Conference on Logic Programming,* pp. 311-337, 1987a.

Shapiro, E.Y., editor, *Concurrent Prolog - Selected Papers,* Vols 1 and 2. The MIT Press, Cambridge, U.S.A., 1987b.

Shapiro, E.Y., OR-parallel Prolog in Flat Concurrent Prolog. *Journal of Logic Programming,* 6(3), pp. 243-267, 1989.

Shen, K., *An investigation of the Argonne model of OR-parallel Prolog.* Department of Computer Science, University of Manchester, 1986.

Sohma, Y., Satoh, K., Kumon, K., Masuzawa, H., and Itashiki, A., A new parallel inference mechanism based on sequential processing. In *Fifth Generation Computer Architectures,* J V Woods (ed.), Elsevier North Holland, IFIP, pp. 3-14, 1986.

Sun, C., and Tzu, Y., The OR-Forest description for the execution of logic programs. *Proc. 3rd International Logic Programming Conference,* In *Lecture Notes for Computer Science,* Vol. 225, pp. 710-717, Springer-Verlag, 1986.

Syre, J. and Westphal, H., *A review of parallel models for logic programming.* Technical Report CA-07, European Computer-Industry Research Center, West Germany, 1985.

Takeuchi, A. and Furukawa, K., Parallel logic programming languages. *Proc. 3rd International Logic Programming Conference,* In *Lecture Notes for Computer Science,* Vol. 225, pp. 242-254, Springer-Verlag, 1986.

Ueda, K., *Guarded Horn Clauses.* ICOT, Japan, TR-103, 1985.

Ueda, K., Making exhaustive search programs deterministic. *Proc. 3rd International Logic Programming Conference,* In *Lecture Notes for Computer Science,* Vol. 225, pp. 270-282, Springer-Verlag, 1986.

ULTRIX-32 Programmer's Manual: Sections 1 and 7. Order Number AA-BG53E-TE, Digital Equipment Corporation, Merrimack, NH, USA, 1987.

ULTRIX-32 Programmer's Manual: Sections 2,3,4, and 5. Order Number AA-BG54E-TE, Digital Equipment Corporation, Merrimack, NH, USA, 1987.

ULTRIX-32 Supplementary Documents: Volume III System Managers. Order Number AA-BG68E-TE, Digital Equipment Corporation, Merrimack, NH, USA, 1984.

Using ARPA Services HP 9000 Series 300. Manual Part Number: 50952-90001, Hewlett-Packard Company, Fort Collins, CO, USA, 1989.

Wang, J., *Towards a computational model for logic languages.* CSM-128, Department of Computer Science, University of Essex, 1989.

Warren, D.H.D., *An abstract Prolog instruction set.* Technical Note 309, AI Center, SRI International, Menlo Park, Ca., 1983.

Warren, D.H.D., *Or-parallel execution models of Prolog.* Technical Report, Department of Computer Science, University of Manchester, England, 1987a.

Warren, D.H.D., The SRI model for OR-parallel execution of Prolog - Abstract design and implementation. *Proc. 1987 International Symposium on Logic Programming,* pp. 92-102, 1987b.

Wise, M.J., Epilog: Re-interpreting and extending Prolog for a multiprocessor environment, In *Implementations of Prolog* (J.A. Campbell ed.), John Wiley & Sons, pp. 341-351, 1984.

Wise, M.J., *Prolog Multiprocessors*. Prentice-Hall, Englewood Cliffs, 1986.

Woo, N.S., and Sharma, R, An And-Or parallel execution system for logic program evaluation. *Proc. 1987 International Symposium on Logic Programming*, pp. 162-165, 1987.

2nd International Workshop on Logic Programming, Universidade Nova de Lisboa, Albufeira, Algarve, Portugal, June 29-July 1, Nucleo de Inteligencia Artificial, 1983.

Wrench, K.L., *A distributed AND/OR parallel Prolog network*. Internal Report, University of Cambridge, England, 1989.

Yang, R. and Aiso, H., P-Prolog: a parallel logic language based on the exclusive relation. *Proc. 3rd International Logic Programming Conference*, In *Lecture Notes for Computer Science*, Vol. 225, pp. 255-269, Springer-Verlag, 1986.

Yashuhara, H., and Nitadori, K., ORBIT: A parallel computing model of Prolog. *New Generation Computing*, 2, pp. 277-288, 1984.