PrologPF: Parallel Logic and Functions on the Delphi Machine

Ian Lewis

Girton College
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

January 1998

Abstract

Prolog PF is a parallelising compiler targeting a distributed system of general purpose workstations connected by a relatively low performance network. The source language extends standard Prolog with the integration of higher-order functions.

The execution of a compiled PrologPF program proceeds in a similar manner to standard Prolog, but uses *oracles* in one of two modes. An oracle represents the sequence of clauses used to reach a given point in the problem search tree, and the same PrologPF executable can be used to *build* oracles, or *follow* oracles previously generated.

The parallelisation strategy used by PrologPF proceeds in two phases, which this research shows can be interleaved. An initial phase searches the problem tree to a limited depth, recording the discovered incomplete paths. In the second phase these paths are allocated to the available processors in the network. Each processor follows its assigned paths and fully searches the referenced subtree, sending solutions back to a control processor. This research investigates the use of the technique with a one-time partitioning of the problem and no further scheduling communication, and with the recursive application of the partitioning technique to effect dynamic work reassignment.

For a problem requiring *all* solutions to be found, execution completes when all the distributed processors have completed the search of their assigned subtrees. If *one* solution is required, the execution of all the path processors is terminated when the control processor receives the first solution.

The presence of the extra-logical Prolog predicate *cut* in the user program conflicts with the use of oracles to represent valid open subtrees. PrologPF promotes the use of higher-order functional programming as an alternative to the use of *cut*. The combined language shows that functional support can be added as a consistent extension to standard Prolog.

Acknowledgements

This work has benefited greatly from the supervision of William Clocksin, and discussion with my colleagues in Austin 316 in the University of Cambridge Computer Laboratory. I am grateful to the University for providing an environment so conducive to creative effort, and hope the UK Engineering and Physical Sciences Research Council feel their funding has been well spent.

The work would not have been possible without the tolerance of my wife Mary and daughter Kitty, and their love and humour have brightened even the darkest days. To them I owe this PhD.

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration. This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other University. No part of this dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification. The text has fewer than sixty thousand words.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

PrologPF is named after **Prolog** in **P**arallel with **F**unctions.

PrologPF is an implementation of a parallel logic language with the following key features:

- The target environment for the compiled binaries is a distributed network of heterogeneous processors with comparatively slow communication links, such as an ethernet or wide-area internet.
- The failure of individual processors during the parallel computation can be accommodated without undue performance penalty
- The language is an extension of sequential Prolog [35].
- OR-parallelism is provided through the use *oracles* to name branches in the search tree to be allocated for distributed search ([28] and section 1.1).
- PrologPF extends the Prolog base language with support for the definition and deterministic application of higher-order functions in a manner consistent with the parallelisation method.

In comparison with other functional logic languages[1], PrologPF uses the efficient but incomplete depth-first search of standard Prolog, and the deterministic eager evaluation of function terms as in functional languages such as Standard ML [55]. Using the comparison drawn by Paulson and Smith in [62], PrologPF is a programming language for *realists* rather than a theorem proving system for *purists*.

[1]Chapter 2 provides a detailed comparison with related work.

The development of the PrologPF system has provided a vehicle for the analysis of scheduling algorithms suitable for the discovery and distributed allocation of oracles. The integration of functions into the logic language provides a means of avoiding the undesirable characteristics of some extra-logical Prolog predicates that would conflict with the parallelisation technique used.

Considerable effort has been made to maintain the portability of PrologPF. The compiler should provide a sound basis for further research such as:

- Improvement of the strategies to be used for oracle distribution.
- Extension of the oracles into the function evaluation trees.

A processor designed to execute the compiled PrologPF programs is called a *path processor*, with the general architecture suitable for creating and following oracles being called the *Delphi Machine*. The previous implementation of the Delphi Machine by Klein [49], further investigated by Saraswat [66], is called in this dissertation *DelphiKS*.

1.0.1 Prolog

The definition of Standard Prolog is contained in [35], and many examples of practical use of the language in [29].

Prolog is a logic language based upon the first-order predicate calculus. A good introduction to the predicate calculus is provided by Lloyd in [50]. A Prolog program is a list of *definite Horn clauses*, i.e. clauses containing exactly one positive literal, these programs called *definite programs* by Lloyd in [50]. A clause with no positive literals defines the *query*. Each clause is a conjunction of literals. A *literal* is a predicate with a list of terms as arguments. A *term* can be a *constant* (i.e. a string or number), a *variable*, or a compound term. Variables are string constants beginning with a capital letter or _. A *compound term* is a string constant with a list of terms as arguments.

A clause containing one literal (which is positive) is called a *fact*. Clauses with more than one literal (of which one is positive) are called *rules*.

Prolog syntax requires that the positive literal appears at the head of the clause, while the negative literals (called the *body*) follow after the symbol "`:-`". The conjunction of the negative literals is represented by "`,`". Clauses end with a full-stop ".".

Comments are surrounded by `/*...*/` or by `%...<newline>`.

Examples of Prolog facts are:

```
a.              % the proposition a
a(b).           % relation a holds for term b
b(c).           % relation b holds for term c
a(b,c).         % relation a holds with arguments (b,c)
a(X).           % relation a holds for any argument term X
```

Examples of Prolog rules are:

```
a :- b.         % asserts a & not(b), i.e. a <= b
a :- b,c.       % a & not(b) & not(c), i.e. a <= b & c
a(X) :- b(X).   % relation a holds for term X if
                      relation b holds for the same term
```

Unification is the process of matching each argument of a subgoal with those in the head of a candidate clause. Genesereth and Nilsson describe the process and provide a pseudo-code algorithm for unification in [39] pages 66-69. The process arrives at a *most general unifier* when the unification is successful and leads to *failure* and subsequent backtracking if unsuccessful. The unifier represents a set of variable bindings which would make the subgoal and the head of the candidate clause identical. These bindings form a context within which the proof process continues.

In solving a query such as `:- a(X,Y),b(Y)` a sequential Prolog interpreter will proceed from left-to-right, finding a solution for each subgoal in turn. To find a solution for a given subgoal (i.e. `a(X,Y)` first), the interpreter will try the program clauses in a top-down order. After a successful unification with the head of a rule, the interpreter will attempt to solve left-to-right the new goal defined by the instantiated body of that rule. On failure of this new subgoal, the top-down search through the program clauses will continue.

1.0.2 Parallelism

Logic languages have potential for faster execution through the exploitation of the available parallelism [71]. The declarative code can be parallelised in several ways, illustrated by the following program fragment:

$$\begin{array}{lll} a(X) & \Leftarrow & b(X,Y)\ \&\ b(Y,Y) \\ b(1,2) & & \\ b(2,2) & & \end{array}$$

A query can be expressed such as $a(Z)$ with the intended meaning that the system should step through the program facts and rules to arrive at values for Z for which $a(Z)$ is provable.

The opportunities for parallel execution within the proof search process include:

AND-parallelism The subgoals $b(X, Y)$ and $b(Y, Y)$ in the body of the rule for a may be searched in parallel to arrive at common solutions for Y.

OR-parallelism The multiple rules for b can be searched in parallel to find solutions for $b(X, Y)$ or $b(Y, Y)$.

Unification parallelism The subgoal $b(X, Y)$ can be solved if suitable values are found for both X and Y. In the selection of a candidate rule, these arguments can be unified in parallel with the formal arguments in the selected rule (e.g. $(1, 2)$).

PrologPF implements OR-parallelism through the use of *oracles* on an extended abstract machine called the *Delphi machine*. Further introduction is given in section 1.1, a review of previous work on the Delphi machine in Chapter 2, and a detailed analysis of the technique in Chapter 3.

1.0.3 Functions

This section provides some background and an introduction to the use of functions in PrologPF. A detailed analysis of the functional support provided in PrologPF is given in Chapter 5.

Functional *reduction* refers to the transformation of a reducible term to a normal form which is considered to be irreducible. When this process is embedded in the code produced by compilation of a functional program, the reduction is called *evaluation*.

The execution of Prolog programs is limited to top-down, left-to-right search with candidate clause matching through unification. The terms given as actual arguments to relations as subgoals are unified directly with the corresponding terms given as formal arguments in the head of the candidate program clause. Thus the Prolog system provides no direct support for the *evaluation* of parameters (with the exception of the `is` relation and related arithmetic terms, see Chapter 5).

However, equivalent relations can be defined representing the required functions in a *flat* form, with the result given as an auxiliary argument. For example the `length` function to produce the length of a list can be defined in Prolog as:

```
length([],0).
length([X|Y],N) :- length(Y,N1), N is N1 + 1.
```

The `length` example illustrates the use of an auxiliary argument to hold the result, and the flat form imposed by the exclusive use of relations, with the exception of the special `is` which evaluates the arithmetic expression given as the second argument. With the definition given above, the relation `length` can be used in a subgoal to produce a variable binding for the length of a given list. The Prolog definition should be viewed in the context of these comments from *Compiling with Continuations*, by Appel [7]:

The beauty of FORTRAN – and the reason it was an improvement over assembly language – is that it relieves the programmer of the obligation to make up names for intermediate results. For example we write $x = (a+b)*(c+d)$ instead of the assembly language:

$$\begin{aligned} r_1 &\leftarrow a + b \\ r_2 &\leftarrow c + d \\ x &\leftarrow r_1 \times r_2 \end{aligned}$$

For comparison with standard Prolog, a definition of `length` as a function in PrologPF would be:

```
fun length([])    = 0;
    length([X|Y]) = 1 + length(Y).
```

With the functional definition in PrologPF, the term `length(X)` can appear anywhere in an argument term to represent the length of the list argument `X`.

The reduction of functional expressions can be defined in terms of the *lambda calculus*[2] In lambda notation, terms are limited to:

Variables. Usually denoted by a constant string such as x.

Constants. Also denoted by constant strings, leaving context to differentiate constants and variables.

Applications. I.e. the application of a function s to an argument t; both s and t may be arbitrary lambda terms. An application can be represented by the simple juxtaposition of the function and argument, e.g. $s\ t$. Application is generally defined to be left-associative, i.e. $s\ t\ u \equiv (s\ t)\ u$.

[2]An introduction to the lambda calculus can be found in [44], while Barendregt provides encyclopedic coverage in [9].

Abstractions. I.e. function definitions in the lambda notation, the function mapping an argument x to a term t being $\lambda x.\ t$.

The application of an abstraction to an argument term relies upon the principle of *substitution*. For example in the term $(\lambda x.\ t)\ a$ the reduction of the application term proceeds with the substitution of the argument a for the variable x in the term t, the result denoted $t[a/x]$. For example, $(\lambda x.\ x\ x)\ a$ reduces to $(x\ x)[a/x]$, i.e. $(a\ a)$.

The x in the above example is referred to as a *bound* variable, representing the formal argument of the lambda abstraction with the extent of its scope limited to the abstraction body. Terms within nested lambda abstractions may contain variables other than those of the immediately enveloping lambda-term, and these variables are said to be *free* within that abstraction. For example, in $\lambda x.(\lambda y.\ x\ y)$ the variable x is said to be free in the term $\lambda y.\ x\ y$. The set of free variables in a term s can be denoted $FV(s)$, and the bound variables $BV(s)$.

Reduction in the lambda calculus is based upon three transformations of lambda terms:

1. **α-conversion.** The constant representing the name of the bound variable in a lambda abstraction can be consistently changed throughout that expression, to any value that is not a free variable in the expression. I.e. $\lambda x.s \rightarrow_\alpha \lambda y.s[y/x]$ provided $y \notin FV(s)$
2. **β-conversion.** The application of a lambda abstraction to an argument term is equivalent to the body of the abstraction with the argument term substituted for the bound variable. I.e. $(\lambda x.\ s)\ t \rightarrow_\beta s[t/x]$.
3. **η-conversion.** A lambda abstraction which applies a term to the bound variable is equivalent to that term, provided the bound variable is free in the term. I.e. $\lambda x.\ s\ x \rightarrow s$ provided $x \notin FV(s)$.

The evaluation of lambda terms is equivalent to the repeated application of α-, β- and η-conversions until there is nothing more to be evaluated. When no more reductions are possible except for α-conversions the term is said to be in *normal form*, and is *irreducible.*

Within this framework there is still considerable flexibility in the selection of the conversion to be applied at each step, and the selection of the subexpression (the *redex*) within the lambda term to be reduced. For example, given:

$$(\lambda x.\ x\ x)\ ((\lambda y.\ a)\ a)$$

then an α-conversion could be applied to either of the lambda-abstractions, resulting in:

$$(\lambda y.\ y\ y)\ ((\lambda z.\ a)\ a)$$

or a β-conversion applied to the second application:

$$(\lambda x.\ x\ x)\ a$$

or a β-conversion applied to the first application:

$$((\lambda y.\ a)\ a)\ ((\lambda y.\ a)\ a))$$

Any implementation of a functional programming language based upon the lambda calculus defines a *reduction strategy.* For the purposes of the functional support in PrologPF the α- and β-conversions are the most significant, with the redex selection for β-conversion being *innermost first* for nested lambda applications.

PrologPF provides a way of naming function abstractions (the `fun` relation) and including lambda abstractions as argument terms (the `lambda` compound term). A full description is given in Chapter 5.

1.1 The Delphi Machine: Background

This section provides an overview of OR-parallel Prolog execution using oracles. A more detailed review of the prior work on the Delphi machine is given in Chapter 2.

1.1.1 The Delphi principle

The Delphi technique for OR-parallel execution of logic programs exploits the following:

1. The search involved in the solution to a given query can be expressed as an *OR-only tree.*

2. Any point in the resultant search tree can be represented by a sequence of integers giving the path to be taken at each internal node leading from the root of the tree to the selected point. The sequence of integers is called an *oracle*

3. The environment at a given point in the search can be recreated by following the associated oracle, and the search continued from there.

The technique can be illustrated by the following example. Figure 1.1 from [28] shows the *AND-OR tree* for the Prolog program given on the left.

The AND-OR tree consists of nodes representing the conjunctive subgoals from the body of the rule, i.e. the AND-node from:

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

p(1).
p(2).

q(1).
q(2).

r(X,X).
```

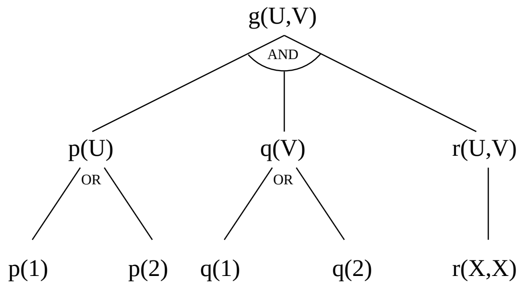


Figure 1.1: Search tree for goal clause g(U,V).

and OR-nodes from the alternate clauses available for the solution of each subgoal. The `p(U)` subgoal transforms to an OR-node with the two children `p(1)` and `p(2)`.

The strict depth-first left-to-right execution strategy of Prolog ensures that a solution to the subgoal `p(U)` is found before execution proceeds with a search for a solution to `q(V)`, and then for `r(U,V)`. This means that the subtree for `q(V)` can be moved and replicated under each leaf-node of the subtree for `p(U)` (Figure 1.2 First Stage). The subtree for `r(U,V)` can then be moved and replicated below each of the resultant leaf nodes, arriving at the OR-only tree in Figure 1.2 (Second Stage).

The OR-parallelism exploited in PrologPF is equivalent to parallel search of subtrees of the OR-only tree in Figure 1.2. If integers are used to label each branch at each OR-node (Figure 1.3) then the sequence of integers leading to a subtree is an *oracle*.

In Figure 1.3 each leaf node is labelled with the associated oracle, and the two major subtrees in this example can be labelled with the oracles *[1]* and *[2]* respectively.

An oracle forms a compact representation of any point within the Prolog search tree for a program with a given query, and parallel computation of the query can be implemented by passing oracles to distributed Delphi machines (also called *path processors*, Section 1.1.2) with an associated *strategy* (Section 1.1.3).

If each n-branch node in the OR-only tree is replaced by a number of binary nodes through the insertion of dummy nodes, then the tree becomes a *binary OR-only tree* with the characteristic that the oracles become binary strings,

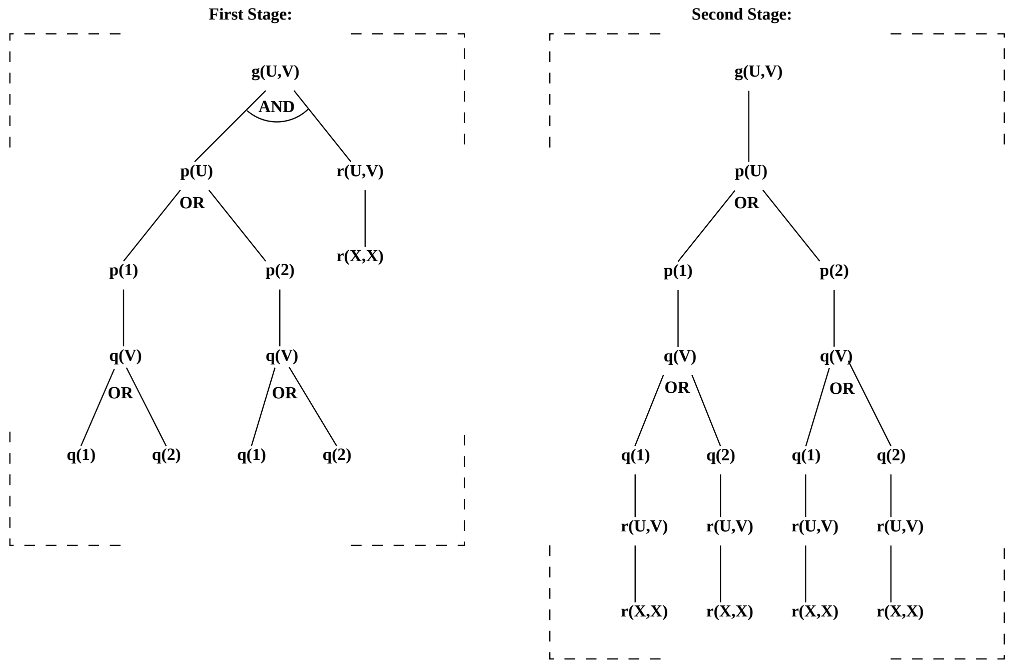


Figure 1.2: Transformation to an OR-only tree.

rather than sequences of natural numbers. This may benefit the generation of oracles using strategies not involving partial search of the proof tree. The strategies evaluated for PrologPF[3] all use partial search to generate oracles, such that the n-ary tree representation is sufficient to describe the operational behaviour. While traversal of the OR-only tree provides an accurate representation of the behaviour of the path processors and defines the interpretation of oracles, it is not necessary to construct the OR-only tree. The current path in the OR-only tree is represented by the path processor's accumulated state, which in turn is defined by the associated current oracle.

1.1.2 Path processors

The distributed *path processors* provide the support for the execution of the logic program, with the abstract machine extended to generate and follow oracles. To accommodate a wide range of distributed execution strategies (Section 1.1.3), the oracle support should include the following:

[3]Chapter 3 provides a detailed discussion of the PrologPF scheduling strategy.

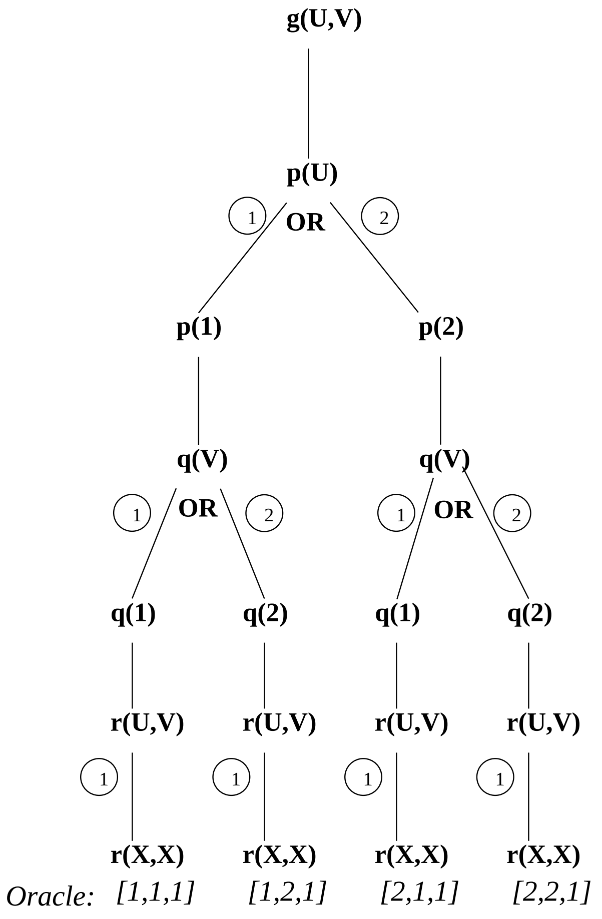


Figure 1.3: OR-only tree with integer branch labels.

1. Given an oracle, the path processor can follow that oracle from the root of the search tree and report the status of that oracle:

   Fail: The oracle resulted in *failure*, either at the end or along its path

   Success: During execution of the oracle, a solution was found. This may occur with a prefix of the oracle, or may have used every integer in the oracle string. The path processor can report the solution with the oracle defining its position in the proof tree.

   Open: The last OR-choice indicated by the last integer of the oracle lead to a successful unification with the head of a rule, such that the execution of the oracle has led to neither *success* or *fail*.

2. The path processor can be asked to search the proof tree within some bound (for example a fixed depth), and to report the open oracles found within that bound.

3. The path processor may be interrupted, when it should pause and report the oracle representing its current search position.

The implementation of the Delphi machine embedded within PrologPF combines the capabilities described in 1 and 2 above, such that the path processor represented by the executing compiled program can:

1. accept a depth bound L as a control parameter which can be zero, any positive integer, or a special value representing infinity

2. follow a given oracle to its end, reporting *success* or *failure* if that occurs

3. continue searching from the end of the given oracle to an incremental depth L, reporting any solutions found within that depth and generating and reporting the open oracles at that depth bound.

These capabilities support a simple scheduling strategy based upon the one-time partitioning of the search tree. The third capability, returning the current oracle on interruption, permits the strategy to be extended to a recursive application of the partitioning algorithm, such that the work of busy processors can be redistributed amongst those that have become idle.

When one PrologPF binary partially searches the proof tree to generate an oracle (or many) for distribution to other processors to follow, the distributed system uses *recomputation.* The Delphi approach trades off the overhead of recomputation with the minimal communications requirement of most successful distributed execution strategies.

1.1.3 Delphi strategies

The object program produced by the PrologPF compiler will execute sequentially on any suitable workstation. Speedup though distributed processing is achieved though the coordinated execution of the same binary on a network of similar workstations used as path processors, with a separate workstation (the *control processor*) controlling the work flow.

The control processor defines the *scheduling strategy* to be used for the allocation of work. As a simple (and very inefficient) example, the control processor could generate oracles at random. These could be sent to randomly selected path processors (with an incremental depth bound L of 0, see Section 1.1.2) until a solution were found.

The support for oracles embedded within PrologPF binaries is sufficient to implement a wide range of strategies. The strategies evaluated in this and previous research[4] include:

- **Non-backtracking strategies:** In these strategies each path processor is used to investigate forwards into the search tree from each assigned oracle, reporting the status to the control processor.

  ***Brute Force*:** [28]. This strategy uses the random allocation of oracles described above

  ***Branch-by-branch*:** [49]. The depth bound L is fixed at 1, and starting at the root of the proof tree, the oracle is extended one digit at a time. I.e. a path processor reports the open single-digit oracles from the root, which are redistributed to the path processors, which report back the open two-digit oracles and so on.

  ***Expanding a Job*:** [28]. As with *branch-by-branch*, except the proof tree is treated as a binary tree, and the oracle is extended a *bit* at a time.

- **Backtracking strategies:** These strategies allow limited backtracking within a path processors after the assignment of an oracle. The objective is to increase the amount of work performed on the path processor before further communication with the control processor is required.

  ***Automatic Partitioning*:** [49]. Each path processor is given G, the number of path processors in the pool, and N the individual processor number. Each path processor uses these numbers to arrive at a unique subtree within the proof tree. At each choice point a given path processor can select a path modulo G with offset N (see Chapter 2 for further detail) and can identify the point at which the path becomes unique to that path processor. The path processor then searches the subtree below this point without constraint.

  ***Reassign-Job*:** [49]. This is a modification to *Automatic Partitioning* to allow path processors encountering *failure* to register with the control processor for further work. Busy path processors are required to poll the control processor (in the Klein implementation) to communicate their current oracle for re-partitioning.

  ***Breadth-first Partitioning*:** [66] and Chapter 3. An initial run takes place with a depth bound L set to generate a suitable number of oracles. These oracles are then all allocated among

[4]For a detailed analysis of related work see Chapter 2.

the available path processors. The path processors follow each assigned oracle, and fully search the subtree below each.

***Partitioning by Selective Sampling*:** [66]. This strategy attempts to improve the effectiveness of the one-time allocation of oracles to path processors by estimating the work beneath each oracle generated in the depth-constrained first phase of breadth-first partitioning. These estimates are used to achieve a more balanced allocation of the oracles to the path processors for subsequent unconstrained search. The work beneath each oracle is estimated by partially searching the subtree (with a limit set on the number of choice-points traversed) and accumulating the number of OR-branches passed during the search.

***Breadth-first Partitioning with Selective Sampling*:** [66]. The final strategy from Saraswat's research has the same goal of improving the allocation of the oracles from an initial breadth-first phase. The method used in this strategy is to fully search the subtree below every other oracle, and use the arithmetic mean of the nodes encountered as a measure of the work associated with the intermediate oracles.

1.2 The Delphi Machine and *cut*

This section gives an overview of the issues surrounding the extra-logical *cut* relation (written '`!`' in Prolog). The topic is covered in detail in Chapter 4. Gupta and Santos Costa analyse the issues with the Prolog extra-logical predicates in AND-OR parallel Prolog in [40].

Figure 1.4 shows the clauses and associated OR-only tree for a program containing *cut*. The procedure for `r` is intended to define a function that maps a first argument `1` to `10`, and any other first argument to `2`:

```
r(1,10) :- !.
r(X,2).
```

A sequential implementation of Prolog will prune away the solution from the second clause for `r(1,X)`. Without the *cut*, there would be two solutions, i.e. {`X = 10, X = 2`}. In an OR-parallel system such as PrologPF, the cut must be communicated at run-time across processor boundaries (represented by the dashed-arrows in Figure 1.4).

In systems implementing the Delphi principle, communication down a path in the tree can be considered to be inexpensive, while between branches (i.e. possibly between path processors) communication may be expensive.

```
g(U) :- p(U), r(U,V), q(V).
p(1).
p(2).
q(1).
q(2).
r(1,10) :- !.
r(X,2).
```

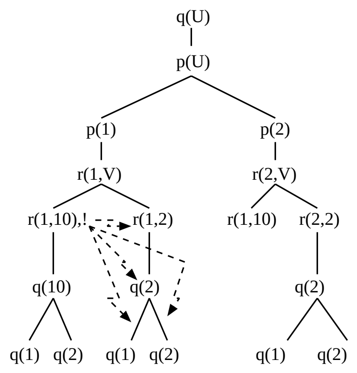


Figure 1.4: Prolog implementation of r(U,V) with cut and transformed tree.

In PrologPF, alternative OR-paths in the proof tree may be executed asynchronously, such that the recognition of the cut is likely to occur after the tree has split further, and communication will be needed between multiple path processors.

A general support for *cut* within the OR-parallel framework of distributed Delphi machines would require a communications system to propagate the *cut* to those path processors searching subtrees that should be pruned. The pruning operation may in effect be a truncation of an allocated oracle, or it may affect the subtree beneath an oracle. Oracle management is thus more complicated. However, the two most critical issues affecting the implementation of general *cut* support within PrologPF are:

1. The *cut* within the program can be expected to be executed many times, generating a great deal of communications traffic if a general distributed support were implemented. PrologPF succeeds in a network of general purpose workstations because the communications traffic is kept to a minimum.

2. The delivery of solutions to a client would have to be delayed until all the path processors have completed, to ensure that all solutions below any *cut* are correctly pruned. A major strength of PrologPF is the efficient delivery of a first solution.

1.3 The Delphi Machine and functions

For a detailed discussion of the functional support in PrologPF see Chapter 5. The developers of the logic language Mercury [69, 43] found that a major requirement for the extra-logical predicate "cut" is to enforce determinism in user code. Deterministic code does not contain any choice points, and the presence of the *cut* thus does not conflict with the OR-parallelism implemented in PrologPF. An example illustrating this is given later in this section.

While *cut* can be used to enforce determinism, *cut* can also be used in nondeterministic relations (those returning multiple solutions). Also a relation containing *cut* may be deterministic with one set of actual arguments, but nondeterministic with another.

The implementation of OR-parallelism using oracles in PrologPF requires that determinism is explicit through the use of *functions* rather than relations containing *cut*. The evaluation of functions in PrologPF is defined to be deterministic. As oracles add no information when the execution tree is linear (i.e. representing a deterministic execution), oracle support can be switched off (see below) while functional evaluation occurs.

The example in Figure 1.5 shows a program similar to that using cuts given earlier in Figure 1.4, but instead uses a function to define `r`. As progression down the tree represents the execution of the program, the function evaluation can be represented as the linear subtree embedded on the right of the proof tree.

```
g(U) :- p(U), q(r(U)).
p(1).
p(2).
q(1).
q(2).
fun r(X) :- if (X=1) then 10 else 2.
```

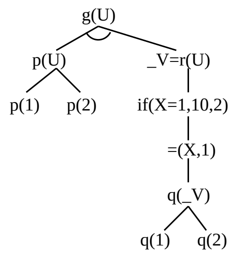


Figure 1.5: Program and search tree for program with function r(X).

Figure 1.6 shows the transformation of the search tree into an OR-only tree suitable for labelling with oracles for allocation to distributed Delphi machines. The linear portions due to the functional evaluation can be seen, and it is clear that the integer labels of the OR-branches can be limited to the alternatives for `p(U)` and `q(_V)`. The path defined by the oracle can be imagined to jump from the last OR-choice to the end of the functional evaluation, and to continue from there.

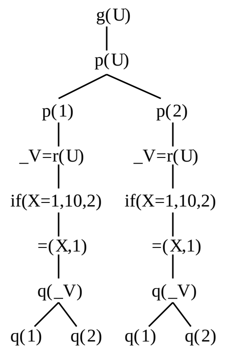


Figure 1.6: Transformation of tree containing r(X) to OR-only tree.

The deterministic evaluation of functions is crucial to the technique of partial suspension of oracle processing used in PrologPF. This ensures that the oracle leading to the start of the functional evaluation branch can equally be said to lead to the branching point of the next OR-choice.

Other functional logic languages, discussed in Chapter 2, aim for completeness in the combined paradigms, providing non-deterministic reduction of functions. The deterministic evaluation of functions in PrologPF was chosen for compatibility with the efficient implementation on the Delphi machine, compensating for the removal of *cut*.

1.4 Research Motivation

Prior work on the DelphiKS implementation of Prolog with the Delphi principle [49, 66] has shown the suitability of the method for OR-parallel execution of pure Prolog programs in distributed systems with relatively high communications costs.

The computing trends exploited by the technique can be expected to continue:

1. The processor performance of generally available computers is increasing faster than the performance of generally available network connections.

2. The number of general-purpose processors available within a general

network environment (i.e. Ethernet or the Internet) is increasing.

Given the success of the technique with pure Prolog programs [66], a compatible extension to Prolog to allow the use of functions should bring the benefits of parallel execution with the Delphi principle to a broader range of problems.

The efficient generation and allocation of oracles within a distributed system is affected by the scheduling strategy used, the overhead of the oracle management techniques, and the communications performance of the network. Further research is needed to provide greater insight into the system behaviour.

1.5 Research Goals

PrologPF has provided an insight into the practical issues of designing a usable environment for the development and parallel execution of un-annotated user programs. In particular, the research goals were:

- to gain further insight into the behaviour patterns of execution algorithms exploiting the Delphi principle
- to extend the Prolog on the Delphi machine with functional features mitigating the removal of *cut*
- to implement a general purpose control system suitable for managing the distributed execution of the path processors
- to test the combined system with a much broader range of Prolog and other code than has been attempted previously

1.6 Contributions

The research documented in this dissertation shows that the Prolog language can be extended with higher-order functions in a manner consistent with the OR-parallel execution of a program with oracles. The combined language can be effectively applied to a broader range of problems than was possible with pure Prolog on previous implementations of the Delphi machine.

The PrologPF implementation is used to study the factors affecting the efficiency of the scheduling strategies, and the influence of the depth parameter L on the breadth-first partitioning strategy is studied in detail. A recursive partitioning strategy supporting the effective redistribution of work amongst the path processors is described.

Chapter 2

Background

This chapter summarises research related to PrologPF, in the areas of parallel Prolog, functional logic, and the prior work on the Delphi machine.

PrologPF is a recomputation-based OR-parallel implementation of Prolog without *assert* or *retract*, and with limited support for *cut*. The language has been extended with the definition and deterministic evaluation of higher-order functions, and the review of related research in this chapter reflects these design choices.

2.1 Parallelism in Prolog

As discussed in Chapter 1, a pure Prolog program can be parallelised by:

- parallel selection of clauses to prove subgoals
- parallel execution of subgoals in the body of a clause
- parallelisation of the unification of multiple or compound arguments

These forms of parallelism are *OR-parallelism*, *AND-parallelism* and *unification parallelism* respectively. Many papers have been written on a wide variety of parallel Prolog implementations. A collection giving a broad coverage of the techniques available can be found in [48].

OR-parallelism is illustrated in the proof tree of Figure 2.1, where the dashed lines surround the subtrees of the proof tree which can be searched in parallel. The conjunctive subgoals `p(U), q(V), r(U,V)` may still be executed sequentially, but the alternative clauses forming the procedures for `p` and `q` may be searched in parallel.

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

p(1).
p(2).

q(1).
q(2).

r(X,X).
```

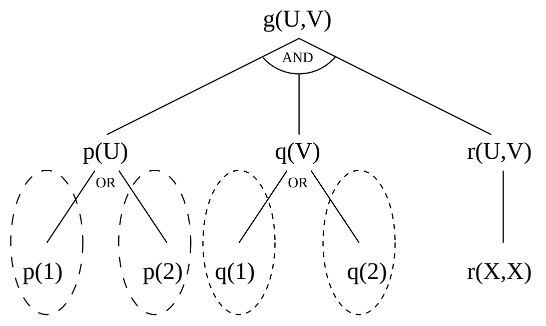


Figure 2.1: OR-parallel execution of goal clause g(U,V).

Similarly, AND-parallelism is illustrated in Figure 2.2, where the subgoals `p(U)`, `q(V)` and `r(U,V)` can be solved in parallel. Support is required to communicate bindings of shared variables between subgoals.

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

p(1).
p(2).

q(1).
q(2).

r(X,X).
```

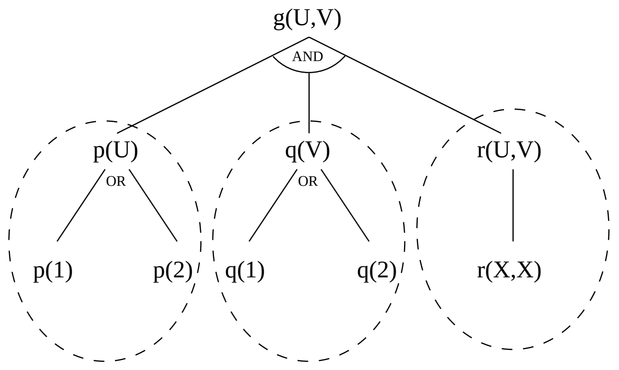


Figure 2.2: AND-parallel execution of goal clause g(U,V).

Systems supporting *AND-OR* parallelism would combine the approaches of figures 2.1 and 2.2 such that all the dashed areas in figure 2.1 could be searched concurrently. The attempted proof of each subgoal involves the unification of the arguments in the subgoal with the arguments given in each clause of the defining procedure. For example, the arguments `U,V` of the subgoal `r(U,V)` will be unified with the arguments `X,X` in the fact defining `r`. In general, the multiple arguments may be unified concurrently, and the unification algorithm itself contains opportunities for parallel execution when compound terms, such as `a(b,X,c(Y))` and `a(Z,b,c(d))`, are unified.

Few systems have exploited the potential concurrency in unification, and the technique is not used in PrologPF. Unification parallelism will not be discussed further in this dissertation.

2.1.1 OR-parallelism

The OR-parallel search of alternate clauses takes place in the context of the variable bindings arising from the search leading to the current choice point. The issue is shown best with the transformed OR-only tree. The dashed areas in Figure 2.3 show the subtrees for `q(V)` for OR-parallel search, in an environment where `p(U)` has already provided the binding {`U/1`}.

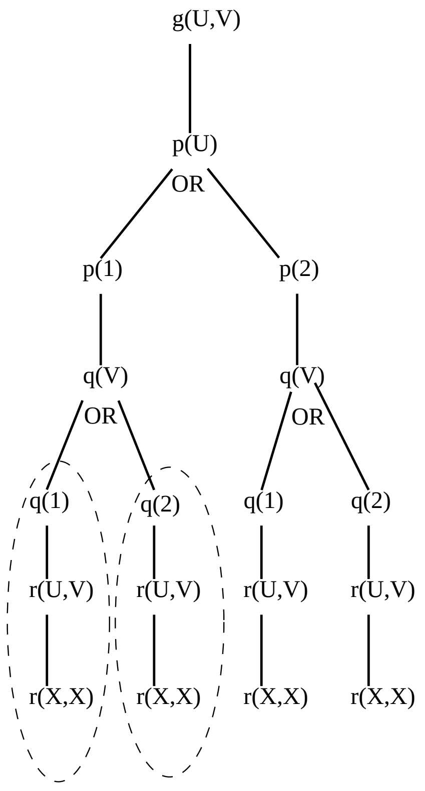


Figure 2.3: Variable bindings in OR-parallel subtrees.

A subgoal `r(U,V)` appears in each dashed area, the first of which contains the binding {`U/1`} from the earlier search and {`V/1`} from the chosen solution for `q(V)`. `r(U,V)` in the second dashed area will be searched in the context {`U/1,V/2`}. Thus with OR-parallel execution the system must ensure that the search continues in the context of the current variable bindings, and that new bindings arising in the OR-parallel subtree must be limited in scope to that search.

Three techniques are commonly used to propagate and limit the scope of variable bindings in OR-parallel systems:

1. **The shared-binding environment model:** a data structure is maintained in memory representing the tree structured binding hierarchy as the search is executed, with each processor building their new bindings and referencing existing bindings higher up the structure. This model is better suited to shared memory computers, illustrated by the implementation of Aurora [52]. A survey of implementations of this type is given by Delgano-Rannauro in [34].

2. **The closed environment family:** at each choice point for OR-parallel execution the environment is copied to each selected processor, which continues by locally extending that context. This technique is suitable for distributed implementation if communications can be minimised, for example by using broadcast to propagate the environment to many processors. The Kabu-Wake approach, described in [54], uses environment copying.

3. **The recomputation family:** the search path to a given choice point for OR-parallel execution is recomputed by the selected processors, such that the environment at that choice point is rebuilt locally in each processor. This technique is suitable for systems with high communications overhead, and the PrologPF system described in this dissertation develops this approach.

The techniques listed above show fundamental design choices in the implementation of the work splitting method. With any of these methods the scheduling *strategy* used to decide at which point the problem should be divided is as important as the technique used to communicate the task. The most simplistic strategy might be to divide the work at every choice point, to as many processors as there are alternate clauses. However, more efficient execution with better load balancing will generally be achieved with more sophisticated strategies. Saraswat studied this issue with DelphiKS in [66], and further discussion for other systems can be found in [4, 11].

PrologPF provides OR-parallelism through recomputation, and the underlying principles and related research are covered in section 2.3.

2.1.1.1 Muse

Muse [4] is named after **Mu**lti-**Se**quential Prolog Engines and is a development of the Multi-Sequential Machine [5]. The system supports OR-parallel execution of full Prolog, with each processor having access to local and shared memory. During execution, the OR-nodes representing the choice points in the search can be either *private* or *shared*. Private nodes are accessible only by the worker which created them. Shared nodes are accessible to all workers searching a subtree beneath that node. Work is divided between worker processors by moving the previous OR-nodes from the private area to the shared area, and incremental copying of the WAM stack (the trail) to the new worker.

The target architecture for the Multi-Sequential machine supports limited broadcast to local memory of each worker [3], so the overhead of copying the WAM stacks to multiple workers is minimised. Muse has been implemented on parallel computers with both broadcast and switched communications support.

The scheduling strategy used in Muse attempts to reduce the overhead of work allocation, with the incremental copying of WAM stacks to new workers and the assignment of multiple choice points to a new worker.

Muse is implemented upon sequential SICStus Prolog [16], and has been shown to have a higher speedup than Aurora (see below) for the same benchmarks [4].

2.1.1.2 Aurora

Aurora [52] is a prototype OR-parallel implementation of Prolog for shared memory multi-processors based on the SRI model [74]. It supports the full Prolog language, thus being able to execute existing Prolog programs without any change. The system was a joint project between Argonne National Laboratories, University of Bristol, and the Swedish Institute of Computer Science.

Aurora uses a storage model in which the path of the search is represented by a group of intertwining WAM stacks [71], with a stack group allocated to each processor. Each choice points creates a branch-point on the stack, and an idle processor can form a branch of the OR-tree emanating from that branch-point. Holes may form in the stack groups when a branch "dies back", i.e. when backtracking fails through the branch-points of the branch. However, that stack group may have been extended with another independent branch, and garbage collection will be delayed until the covering branch is completed.

In searching the branches from that choice point, multiple processors can produce independent solutions, i.e. different but valid variable bindings. Thus bindings cannot be stored as values in the logical variables, and a *binding array* is used per processor. The binding array is essentially a software cache of variables and their values, for exclusive use by the associated worker processor.

The Aurora systems is implemented using SICStus Prolog [16] and has provided a platform for the evaluation of multiple scheduling strategies [11]. The scheduler determines how tasks should be allocated to idle worker processors and synchronises the access to the shared nodes nearer the root of the search tree.

2.1.1.3 Kabu-Wake

The Kabu-Wake model [54] is based upon environment copying with selective backtracking to allow processors to compute alternate paths.

A processor computes sequentially until it is interrupted with a request for work from an idle processor. The busy processor (called the *parent*) suspends its computing when the request is received, sends a copy of its environment to the idle processor, and then resumes. Part of the splitting procedure requires the parent to temporarily backtrack to the splitting choice point, so that the more recent variable bindings from the choice point are undone. In order to recognise the validity of the bindings, the system uses an incremental time-stamp in each variable cell.

The model leaves open the specification of the algorithm for the selection of a suitable parent by an idle processor. The response of a parent to an interruption is immediate, without optimisations to improve the task granularity. Load balancing is performed by the selection of the parent processors by those which are idle. Efficient performance would require the copying be minimised by targeting problems with well balanced search spaces [34].

2.1.1.4 OPERA

The OPERA project [17] was inspired by the Kabu-Wake model (see above) with the principle that the complete state of a busy processor is transferred to an idle one to effect work sharing. The target architecture is similarly distributed processors with a high-performance communications network providing node-to-node connectivity. The implementation of OPERA on a dynamically reconfigurable array of Transputers is optimised for a system with relatively long connection setup times ($\geq$ 250 μs) but an efficient matrix block transfer performance. Each point-to-point connection can transfer data at between 500Kbps and 1Mbps (b=byte). DMA is used (Direct Memory Access) to move the data into and out of the processor memory so that processing can continue concurrently with the data transfer. Most importantly, a crossbar switch system is used to implement the network so that many transfers can take place in the network concurrently. The relatively long communications setup time means that short data transfers are relatively inefficient, precluding the use of stack sharing models as in Aurora (see above).

A multi-sequential approach is used: each processor executes a complete Prolog engine based upon an extended WAM. The stack data structures are modified to improve the efficiency of the copying operation. Choice points are managed in a separate double-linked list, rather than being intertwined with the clause activation records as in a standard WAM. This separation is similar to the technique used in Muse (see above), and improves the efficiency of work splitting. Variable bindings on the trail stack are time-stamped. Work splitting at a given choice point would require all variable bindings that had occurred after that choice point be unbound. The time-stamps (as in the Kabu-Wake model) mean that the copy process need not thread though the trail stack to unbind these variables, but can simply compare the time-stamp with that of the choice-point. To minimise further the amount of stack copying, the prototype always splits with work of an active worker at the topmost choice point (i.e. nearest the root), such that the stacks to this point are as short as possible.

As the cost of task creation on an idle processor is relatively high, involving the copying of the state of the active processor, effective scheduling in OPERA is important [17]. The scheduler has to consider the export and import time of the active and idle processors compared to the expected time for the search of the current subtree to complete. The scheduler should ensure that the active worker, in passing choice points to an idle worker, keeps enough work to remain active after the initialisation of the new task. After consideration of alternatives, a scheduling model with a hierarchy of scheduling processes was used, with *spy* processes on each worker processor to estimate the workload of the active workers. The workload estimate is performed dynamically, with the simple heuristic being used of the number of choice points being held by the worker.

Good speedups have been achieved with the prototype up to a maximum of 16 processors. Further developments are aimed at reducing the task creation overhead through the use of incremental copying.

2.1.1.5 ANL-WAM

The ANL-WAM was an early experimental implementation of OR-parallel Prolog at Argonne National Laboratory on a shared-memory multiprocessor (a 20-cpu Encore Multimax). The principles evaluated using this system [37] were used in the subsequent development of Aurora (see above).

As with Aurora, a hash-table structure was used to cache the multiple variable bindings arising from OR-parallel execution of alternate choice points. With ANL-WAM, starting a new worker process involves giving the worker access to the variable bindings created so far, and the creation of a new hash table to store the new variable bindings resulting from the allocated branch. As shared memory was used, the copying process could be limited to the headers of the hash table with pointers to the existing shared nodes. The allocation of work to new workers was thus efficient, with the design decision taken to trade this against contention for subsequent access to shared variables.

The scheduling algorithm used in ANL-WAM created a fixed number of worker processes to be assigned to branch points as they were created. On finishing a branch, the process will seek more work to do from a dispatching pool. A graphical display tool was created to play back a trace log showing the growth of the search tree and the allocation of branches to worker processes. The tool was used to improve the dispatching algorithm.

Results were produced from ANL-WAM [37], showing effective speedups for some problems up to a maximum of 16 worker processes.

2.1.1.6 Boplog

Boplog [72] is a multi-sequential OR-parallel Prolog design implemented on the BBN Butterfly Parallel Processor, which is a multi-cpu shared memory design. The memory consists of segments local to each processor, which can be accessed remotely by all other processors. Each processor's address space is defined locally, such that an address of a word on a remote cpu may be different to different processors. The Boplog implementation attempts to optimise the use of the segmented memory.

To support parallel execution, the design makes extensive use of shared data structures rather that structure copying, as the non-local memory access time (6.3μs) was considered reasonably fast compared to the local memory access time of 1.35μs. The resulting slower access time to shared data and less efficient reclamation of heap and stack space, which cannot be released until no other processors need access to it, are traded for less time for copying and less memory for redundant or unused data [72].

In Boplog, variable bindings are time-stamped and stored in a doubly-linked list to improve the efficiency of the work reassignment. Scheduling is achieved by idle processes obtaining more work from busy processes, selecting the untried branch nearest the root of the search tree. The early analysis of Boplog's runtime behaviour suggested that work was reassigned on average every millisecond, with the allocation typically involving the transfer of 100 bytes of data.

2.1.2 AND-parallelism

With the AND-parallel execution of a Prolog Program, the conjunctive subgoals in the body of a clause are solved concurrently, while the alternative clauses in a procedure are tried sequentially. Conery and Kibler in [31] suggest the model can be further divided, according to the handling of shared variables, as follows:

Independent AND-parallelism: Even when the subgoals share variables they are solved independently. After all solutions are found, the shared variables are tested for consistency.

Dependent AND-parallelism: Also called stream-and parallelism, subgoals with shared variables are executed dependently, that is they interfere with one another. The word *stream* is used to represent the flow of bindings from one AND-parallel process to another.

Restricted AND-parallelism: Subgoals sharing no variables are executed in parallel, while subgoals sharing variables are executed sequentially.

The communication of solutions (i.e. variable bindings) between AND-parallel subgoals is illustrated in Figure 2.4 where the three subgoals for `p`, `q` and `r` are represented by the processes A, B, and C respectively.

The following sections summarise implementations of the AND-parallel logic computation model. PrologPF is based upon the purely OR-parallel Delphi machine, but an effort by Wrench [76] to extend the machine to support both AND- and OR-parallel execution is summarised in section 2.3.

2.1.2.1 Parlog

Parlog is a stream AND-parallel logic programming system in which the logic variables can be thought of as channels, down which partial results are

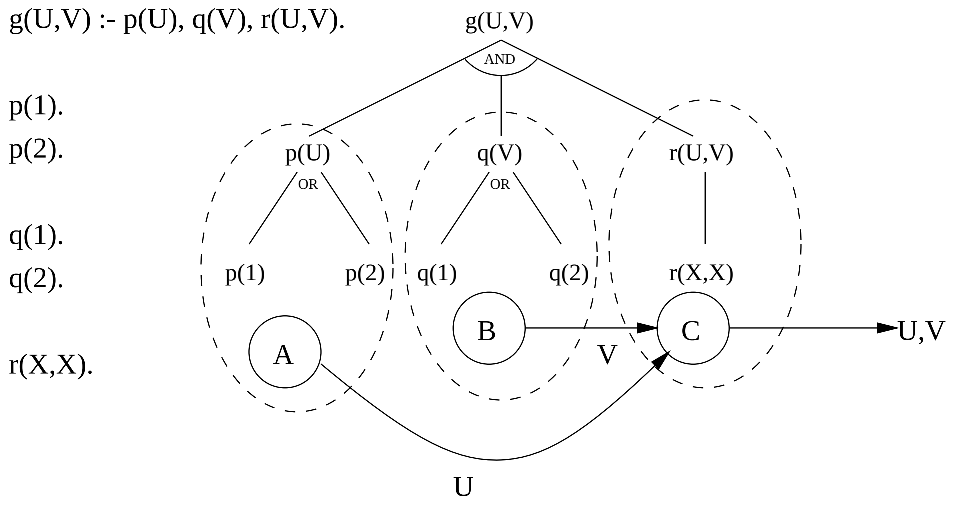


Figure 2.4: Communication of bindings in dependent AND-parallelism.

sent between literals that are executed in parallel [24]. The model is suited to implementation on a dataflow architecture computer, or a conventional multiprocessor with shared memory.

The language implemented in Parlog is that of guarded Horn clauses, built upon the syntax and semantics developed in [23]. Procedures are annotated with *mode* information, specifying which logical variables should be inputs and outputs to each procedure. The *guards* are goals added to each clause so that the form of clause selection is *committed choice*, i.e. only one clause will be selected for which the guard literals evaluate to *true*. Parlog will only ever find one solution to a query. At the time of the parallel guard evaluation, all guard literals must be *ground*, i.e. contain no variables. Both serial and parallel forms of connectives can be used in the definition of the goal sequence in the body of a clause. "`&`" implies sequential left-to-right execution of the goals, while "`,`" implies the goals can be executed in parallel. A later extension to Parlog allowed similar annotation of the clauses in a procedure, where "`.`" permits OR-parallel search, while "`;`" implies top-down sequential search.

The treatment of variables in Parlog in optimised for stream AND-parallelism, with the *don't care* parallelism (i.e. the commitment to the first goal with a successful guard) limited to committed choice non-determinism. The OR-parallel execution is provided though all-solution operational model using set expressions.

2.1.2.2 Concurrent Prolog

Like Parlog, Concurrent Prolog in based upon the stream AND-parallel model and associated committed choice language proposed in [23]. The system has been simulated in Prolog, with the proposal that it is best suited to multiprocessor dataflow architecture machines [68].

Concurrent Prolog does not require the guard literals to be ground at the time of evaluation, such that clause selection (and commitment) does not just rely upon successful evaluation of the guard sequence as in Parlog. The evaluation of the guards must also return a set of variable bindings. The committed choice nature of the clause selection, with the resultant single solution to each goal, means that for many general logic programs the system may fail to find a solutions. For example [75],

```
simple(X) :- p(X),q(X).
p(1).
p(2).
q(1).
q(2).
```

The Concurrent Prolog query `solve(simple(X))` may commit to the solution `X=1` for `p(X)` and subsequently fail the goal `q(X)`. In a nutshell, both Parlog and Concurrent Prolog have adopted a semantics markedly different than that of sequential Prolog.

2.1.2.3 Delta Prolog

Delta Prolog is a logic programming language extending Prolog with constructs for sequential and parallel composition of goals, interprocess communication and synchronisation, and external non-determinism [32]. The language is optimised for execution on distributed machines, and makes extensive use of the concepts of communicating sequential processes (CSP) developed by Hoare in [45].

As with CSP, parallelism is made explicit in Delta Prolog through the use of a *split* operator `//` where goals S_1//S_2 are to be evaluated in parallel. Channels for communication between goals are established though the use of *event goals*, with `X!chan` being considered to send the value of `X` along channel `chan`, to be received by a complimentary subgoal `Y?chan` in a concurrently executed goal. To be an acceptable candidate to receive the value, the arbitrary terms `X` and `Y` must be unifiable, and the atom naming the channel the same in both the transmitting and receiving event goals. The event goals can be *guarded* though the use of associated goal sequences with the syntax `X?chan:G` where `G` is the goal sequence which must evaluate to `true` for the communication event to be accepted. Non-deterministic acceptance of a communication event is provided though the definition of *choice goals*. These goals have the form $\texttt{A}_1\texttt{::A}_2\texttt{::...::A}_n$ where each $\texttt{A}_i$ is of the form `H,B` with `H` an event goal and `B` the (possibly empty) body of the alternative clause.

The system provides efficient support for the communication of values though the use of event goals. Non-deterministic evaluation of goals requires the implementation of distributed backtracking. The prototype implementation supports backtracking in some simple programs, but this is an area of ongoing research.

2.1.2.4 EPILOG

EPILOG [75] is wholly based upon the dataflow model of computation. In place of Prolog's depth-first left-to-right evaluation strategy the EPILOG model, by default, evaluates all clause-body literals in parallel, that is performs breadth-first execution. All solutions to a query are found in parallel, and *back-unification* (the propagation of unifiers from subgoals back up to higher level goals) is used to be *equijoined* with other partial solutions to be propagated to still higher level goals.

The process is illustrated for the query `ans(X,Y,Z)` in figure 2.5.

EPILOG would be overwhelmed with data if no mechanisms were provided to reduce the combinatorial explosion of data transfers arising from the breadth-first nature of the parallel execution. Fixed sequencing constructs are provided to reintroduce left-to-right evaluation of clause-body literals and to order the clauses in a procedure. In addition, *mode* information on variables can be specified, and *thresholds* can be specified giving the number of arguments to be ground before a clause will be executed.

2.1.3 Other forms of parallelism in Prolog

As was mentioned in Chapter 1, the unification algorithm used in Prolog to match a subgoal with a suitable clause head contains opportunities for parallel execution.

Parallel unification of multiple arguments: With a subgoal `p(a,b,c)` and a clause `p(X,Y,Z) :- ...` each argument can be unified in parallel, arriving at the unifier `X/a,Y/b,Z/c`. Where variables are repeated in the clause head, a communication mechanism must be provided to synchronise the shared binding.

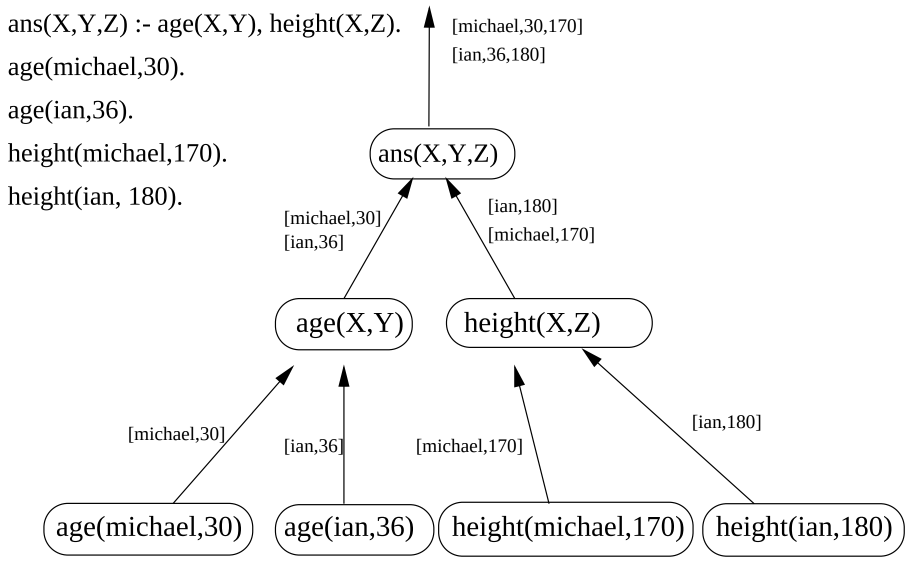


Figure 2.5: Dataflow communication of bindings in EPILOG.

Parallel unification of compound subterms: Each argument to a relation may be a compound term with a tree internal representation. Different branches of the tree may be unified in parallel with corresponding elements of the argument in the clause head. This technique is a generalisation of the one given above.

Effort into the concurrent execution of the unification algorithm has been limited, and in PrologPF and other OR-parallel Prolog systems the unification algorithm is sequential.

2.2 Functional Logic

The integration of functional and logic programming languages can be approached from either a functional or logical starting point although both techniques lead to similar operational principles [41]. As the primary foundation for PrologPF is logic programming, the existing research listed here emphasises that approach.

A survey of the field giving an introduction to the alternative approaches can be found in [12], and a more recent summary with an emphasis on narrowing [64] with application specific abstract machines can be found in [41].

2.2.1 Functions as deterministic Prolog procedures

The eager evaluation of a function of N arguments can be replaced with the execution of a Prolog goal of $N + 1$ arguments, where the additional argument is a logical variable to hold the result. For example the function `factorial(N)` can be replaced with the relation `factorial(N,F)`:

```
factorial(1,1).
factorial(N,F) :- N > 1,
                  N1 is N - 1,
                  factorial(N1,F1),
                  F is N * F1.
```

The advantage of this approach is that the simple syntax and semantics of standard Prolog can be retained for functional programming as well as non-deterministic logical programming for which Prolog was designed. The disadvantages include:

1. **Higher-order functional programming:** functions are not treated as first-class data items in Prolog. For higher-order application of functions the programmer must adhere to arbitrary programming conventions and use extra-logical relations such as `call` to use functions as arguments and results. Effort has been applied to retaining the relational definitions of functions but adding higher-order support to Prolog, particularly through the use of `call/N` [69, 58] and `apply/3` [58]. These techniques are compared with PrologPF in Chapter 5.

2. **Flat programming style:** the flat syntax of standard Prolog means that all intermediate functional results must be given a name. This requirement has been likened to assembler [7]. To reduce the problem, Prolog supports nested arithmetic expressions as the second argument to the special `is` relation, but this support is arbitrarily limited to this special use.

3. **Use of *cut*:** The deterministic evaluation of functions typically requires the use of guard conditions in the definition of the alternate clauses in the Prolog procedure (see `N > 1` in the `factorial` example above). For a procedure with many clauses, the guard conditions can become unwieldy, such that the use of *cut* simplifies the definition of the sequential algorithm, excluding subsequent clauses from providing alternative solutions. The use of *cut* introduces considerable complexity in the OR-parallel execution of Prolog programs, and the issue is covered in detail in Chapter 4.

4. **Execution efficiency:** the execution model for the deterministic eager evaluation of functions lends itself to an efficient implementation compared to the unification and backtracking requirements of a Prolog program. The compile-time analysis of logic programs to recognise deterministic modes of execution is a topic of current research in systems such as Mercury [43]. The use of *cut* does not imply determinacy (see Chapter 4). Use of a syntax for functions other than that deterministic procedures in standard Prolog can render explicit the requirement for deterministic evaluation.

2.2.2 Term evaluation

The most straightforward approach to adding functions to Prolog is to require the arguments to be fully instantiated before reduction is attempted. The operational semantics of Prolog can be maintained and the responsibility for this requirement placed on the programmer, e.g. in the standard *is* predicate. Alternatively, function evaluation can be deferred until this condition is met. The technique of deferral is called *residuation*, see [2].

A simple example from [53] may serve to illustrate the principle:

```
length([], 0).
length([X|Xs], N + 1) :- length(Xs, N).

:- length([a,b,c], 5).
no

:- length([a,b,c,d,e], 5).
yes

:- length([a,b,c], L).
L = 3

:- length(List, 5).
List = [_,_,_,_,_]

:- length(List, L).
List = [], L = 0;
List = [_], L = 1;
List = [_,_], L = 2;...
```

In the above example, `N + 1` is a functional term providing a natural expression of the problem with more generality than that provided by `is`. This is illustrated by the last two examples, where with the Prolog operational semantics non-ground functional terms (i.e. $N+1$) will be encountered during execution. In implementations providing residuation (e.g. Le Fun [2], GAPLog [53]), the function call (+) will be delayed. In the case of arithmetic, unification of two terms t_1, t_2 reduces to solving the equation $t_1 = t_2$. If further constraints are imposed upon the arguments, namely [53]:

- *Equivalent arguments.* t_1 and t_2 are equivalent if and only if evaluation of all their ground subterms makes them identical, and unification succeeds with a null unifier.
- t_1 *or* t_2 *is a variable* X. If the other term t does not include X (occurs check) then unification succeeds with the mgu $\theta = \{X/t\}$.

then we approach the limitations of the predefined Prolog predicate `is`, supporting calls such as `X is 3 + 4` and `7 is 2 + 5` [1].

Instantiated term evaluation allows external functional procedures to be used in Horn clauses, and does not require the definition of the functions in a common language. As the determinism of the functions is explicit, the programs can be executed more efficiently than within the general execution mechanism of the logic programming environment.

While residuation ensures the function calls are only made when sufficiently instantiated, the procedure is essentially incomplete and does not allow for function inversion.

2.2.3 Mode and determinism declarations for relations

Many Prologs include support for mode declarations for system- and user-defined relations, for example the SICStus list library relation to return the maximum member of a list [16]:

```
maxlist(+,?)
```

indicates that the first argument to `maxlist` must be fully instantiated (i.e. ground) before the call, and the second argument can contain zero or more variables (i.e. be ground or non-ground). Thus permitted calls include:

```
maxlist([1,2,3],X)
```

[1] In fact with Prolog's *is* only the right-hand argument is evaluated, so `2 + 3 is 4 + 1` fails. The built-in arithmetic predicate '=:=' will evaluate the arithmetic expressions on both the left-hand and right-hand sides, each side must be ground and `Z =:= 2 + 3` fails.

which will succeed with the substitution {`X/3`} and

```
maxlist([1,2,3],2)
```

which will fail.

The mode declaration provides an opportunity for the Prolog compiler to produce more efficient code, as choice points can be eliminated and the unification of parameters need not be as general. Most implementations of Prolog (e.g. SICStus Prolog Version 3) perform *no* optimisations based on the mode statements provided by the programmer, and the information is treated as a comment.

Other logic languages, in particular Mercury [69], make extensive use of the mode information to generate efficient code. In the Mercury syntax, the `maxlist` relation would have two modes:

`mode maxlist(in,out)` and `mode maxlist(in,in)`

Note that for every mode of a predicate in which an argument is *produced* (mapped from free to bound) there is another mode for that predicate in which the argument is *consumed* (mapped from bound to bound), and similarly arguments ignored (mapped from free to free) have another mode with the argument mapped from bound to bound. These additional modes are referred to in [69] as *implied modes.* In addition, Mercury can annotate mode declarations with their intended determinism, with tags of `det, semidet, or nondet` to indicate that calls of the given mode have exactly one solution, zero or one solutions, or zero to many solutions respectively. The Mercury compilation process transforms a logic program to C, and the current implementation generates separate code for each declared and implied mode of each predicate.

This meta-logical information enables the compiler to perform significant additional error checking and to generate efficient code for each mode. Inline code can be generated for some relations and for certain instances of unification, for example instances of `X = Y` where one of X and Y is input (i.e. ground $\rightarrow$ ground) and the other is output (free $\rightarrow$ ground). Deterministic relations are transformed directly into C code for an efficiency comparable with that of imperative languages, while relations with the mode *semidet* are transformed into deterministic C code returning a success or fail indication. Non-deterministic modes are supported with the simulation of a virtual machine similar to the WAM [1]. Execution of some simple deterministic and nondeterministic benchmarks (translated from Prolog) suggests an improvement in efficiency from two to five times with Mercury's execution algorithm.

The definition of functions in logic programming languages has considerable overlap with the definition of deterministic (or semi-deterministic) relations, and the performance gains from more efficient execution of deterministic code should be similar in each case (i.e. substantial). It remains to be seen whether the use of mode declarations or function definitions are the clearest way of expressing this determinism.

2.2.4 Predicates as set-valued functions

This approach proposed in [63] has been investigated further in [19] to address the incompleteness of Prolog's depth-first execution strategy. The clauses of the logic program together with an input/output mode for the goals are transformed into a system of mutually recursive definitions of set-valued functions (SVF's). The transformation technique has the restriction that only ground bindings are permitted for the output variables of the goals. The evaluation of the set-valued functions is performed essentially in a top-down, depth-first fashion, and critical to the implementation in [19] is the provision of a *functional environment* to implement a *memo-structure* such that loops in the functional code can be recognised and those calls suspended. The use of moding, translation, and the operational principles can be illustrated with a simple example:

`source program:`	$path(X,Y) \leftarrow arc(X,Y)$
	$path(X,Y) \leftarrow path(X,Z), arc(Z,Y)$
`mode:`	$path^{+-}$
	arc^{+-}
`target SVF:`	$path^{+-}x = arc^{+-}x; (path^{+-}x)\{arc^{+-}\}$

It is assumed that *arc* is defined by ground unit clauses. Set union is written as ";" and the construct $E\{F\}$, where E is a set-valued expression and F is a set-valued function, denotes the set formed by applying F to all the elements of E and taking the union of the resulting sets. The definition of the target function includes a loop (with $path^{+-}x$ on both the left- and right-hand sides) and a standard functional evaluator would loop even though the definition of arc might represent a tree so the set associated with $path^{+-}$ is finite. Hence the memo-structure.

This approach addresses the issue of completeness in depth-first SLD-resolution by providing an alternative operational semantics and constraining the use of logical variables. However, the limitations of the current approach are incompatible with the goals of the proposed research.

2.2.5 Predicates as Boolean functions

An example of this approach can be found in the language Escher [51], which is essentially a functional language founded upon higher order logic based on Church's simple theory of types. Predicates are regarded as functions which map into the domain of type Boolean, and must be *moded*.

In common with many functional languages, the following limitations apply to function definitions:

- *Constructor-based.* User declared functions are either *free* or *defined*. A function is defined if it appears as the outermost functional symbol on the left-hand side of a rewrite rule. These rules define an equality on terms with a direction of the rewrite, which in Escher and most functional languages are always left to right. *Free* functions are irreducible and can equally be viewed as *constructors*. With $\mathcal{F}$ the set of defined functions and $\mathcal{C}$ the set of constructors, $\mathcal{F} \cup \mathcal{C}$ is the program and $\mathcal{F} \cap \mathcal{C} = \emptyset$. If $l \Rightarrow r$ is a rule, then all functions in l except the outermost must be in $\mathcal{C}$. This precludes expression of equalities such as $append(append(a, b), c) = append(a, append(b, c))$.

- *Left linearity.* No variable appears more than once in the left-hand side of a rule.

- *Free variables.* If $l \Rightarrow r$ is a rule and $Vars(t)$ is the set of variables appearing in term t, then $Vars(l) \supseteq Vars(r)$. Thus all variables in the body (r) of the rule must be bound.

- *Non-ambiguity.* If the outermost function symbol of a term t is $Outer(t)$ and $\mathcal{R}$ is the set of rewrite statements of the form $l_i \Rightarrow r_i$ defining function f, i.e. $\forall(l_i \Rightarrow r_i \in \mathcal{R}) : (Outer(l_i) = f)$, and $Mode(f)$ is the mode definition for f then exactly **one** statement in $\mathcal{R}$ must match any call under $mode(f)$.

In addition in Escher, where $mode(f)$ specifies a `NONVAR` argument, the corresponding term t in the call must contain no variables ($Vars(t) = \emptyset$), and where $mode(f)$ allows an argument to be input or output (represented as _), the corresponding argument in the function definition must be a variable.

The principles can be illustrated with an example modified from [51]:

```
FUNCTION Nil:   Unit -> List(a);
 Cons:  a * List(a) -> List(a);
 a,b,c: Unit -> Item.

FUNCTION Split: List(a) * List(a) * List(a) -> Boolean.
MODE     Split(NONVAR,_,_).
 Split(Nil,X,Y)       => (X = Nil) and (Y = Nil).
 Split(Cons(X,Y),V,W) =>
    (V = Nil) and (W = Cons(X,Y) or
     SOME [Z] ((V = Cons(X,Z)) and Split(Y,Z,W)).

FUNCTION Append: List(a) * List(a) -> List(a).
MODE     Append(NONVAR,_).
 Append(Nil,X)       => X.
 Append(Cons(U,X),Y) => Cons(U,Append(X,Y)).
```

The computational model is that of "rewriting" rather than theorem proving, and the call `Append([a,b],[c])`, with sugaring for `Cons`, will be repeatedly rewritten:

$$\begin{array}{l}
\underline{\texttt{Append([a,b], [c])}} \\
\qquad\quad \Downarrow \\
\texttt{[a | }\underline{\texttt{Append([b], [c])}}\texttt{]} \\
\qquad\quad \Downarrow \\
\texttt{[a, b | }\underline{\texttt{Append(Nil, [c])}}\texttt{]} \\
\qquad\quad \Downarrow \\
\texttt{[a,b,c]}
\end{array}$$

This form of rewrite via function calls is straightforward. However, an example with the function `Split` illustrates the need for additional rewrite schemas:

$$\begin{array}{l}
\underline{\texttt{Split([a,b],X,Y)}} \\
\qquad\quad \Downarrow \\
\texttt{(X=Nil and Y=[a,b]) or} \\
\texttt{SOME [Z] (X=[a|Z] and }\underline{\texttt{Split([b],Z,Y)}}\texttt{)} \\
\qquad\quad \Downarrow \\
\texttt{(X=Nil and Y=[a,b]) or} \\
\texttt{SOME [Z] (X=[a|Z] and ((Z=Nil and Y=[b]) or} \\
\qquad\qquad\qquad\qquad\quad \texttt{SOME [Z'] (Z=[b|Z'] and }\underline{\texttt{Split(Nil,Z',Y)}}\texttt{)))} \\
\qquad\quad \Downarrow
\end{array}$$

```
(X=Nil and Y=[a,b]) or
SOME [Z] (X=[a|Z] and ((Z=Nil and Y=[b]) or
                       SOME [Z'] (Z=[b|Z'] and (Z'=Nil and Y=Nil))))
           ⇓
(X=Nil and Y=[a,b]) or
SOME [Z] (X=[a|Z] and ((Z=Nil and Y=[b]) or (Z=[b] and Y=Nil)))
           ⇓
(X=Nil and Y=[a,b]) or
(X=[a|Z] and Z=Nil and Y=[b]) or
(X=[a|Z] and Z=[b] and Y=Nil)
           ⇓
(X=Nil and Y=[a,b]) or
(X=[a] and Y=[b]) or
(X=[a|Z] and Z=[b] and Y=Nil)
           ⇓
(X=Nil and Y=[a,b]) or
(X=[a] and Y=[b]) or
(X=[a,b] and Y=Nil)
```

This example illustrates the use of rewrite rules for user-defined functions, logical and existentially quantified expressions. The Escher system has many rewrite schemas with the general process referred to in [51] as *simplification*. Examples include:

$$\begin{aligned}
\mathit{False} \wedge A &\Longrightarrow \mathit{False} \\
(A \vee B) \wedge (A \vee C) &\Longrightarrow A \vee (B \wedge C) \\
\exists x_1 \ldots x_n (A \wedge (x_i = T) \wedge B) &\Longrightarrow \exists x_1 \ldots x_{i-1} x_{i+1} \ldots x_n (A\theta \wedge B\theta) \\
& \text{(where } \theta = \{x_i/T\} \text{ and} \\
& \quad x_i \text{does not occur in } T)
\end{aligned}$$

The approach can provide great flexibility but performance is an open issue, with the implementation needing to search large and complex terms to find suitable redexes, and selecting from a choice of over 100 rewrite schemas to be applied. Given the functional foundations of the technique, the implementation in Escher has statements as equations, in the functional style, rather than implicational formulas in the logic programming style. Also there is no *explicit* concept of non-determinism, which instead is represented implicitly by disjunction. Computations return all answers, and failure is represented by returning $False$.

The implementation of Escher provides a complete search process without use of non-logical features such as *cut*, but the rewriting semantics equate to the delivery of all solutions at each stage of the proof, with the corresponding cost in space and time.

2.2.6 Resolution extended to equational systems

This approach provides the direct integration of functions into the logic language, permitting program clauses defining the equality predicate. Whereas the equality predicate "=" is predefined in Prolog as if with the rule `X = X`, functions can be defined by admitting new clauses for "=" and extending the Prolog operational semantics to include term rewriting, resulting in an amalgamated language referred to as *logic programming with equality.*

The general procedure is to unify each non-variable subterm of the goal with the left-hand side of an equality rule and replace the subterm with the instantiated right-hand side of the rule, until the sub-term is ground. The process is referred to as *narrowing* [65]. A detailed analysis can be found in [47], annotated with examples in [41]. Extended unification algorithms are surveyed in [36]. In summary, given:

a set of function symbols F

a countable set of variables V

a *term* is either a variable $\in V$ or of the form $f(t_1, \ldots, t_n)$, where $f \in F$ and $t_1 \ldots t_n$ are terms

a set $\mathcal{T}(F, V)$ of all terms over F and V

a set $Vars(t)$ of the variables in term t

term t is *ground* iff $Vars(t) = \emptyset$

if u is a non-variable subterm of t at position p, then $t|_p$ denotes u and $t[u']|_p$ denotes the result of replacing the subterm $t|_p$ by the term u'

a *substitution* θ is a mapping from V to $\mathcal{T}(F, V)$. $\theta(t)$ represents the term obtained by replacing the variables of t with their substitutes in θ

a *rewrite rule* is of the form $l = r$ with $Vars(l) \supseteq Vars(r)$

a program is a set of rewrite rules $\mathcal{R}$

term t is *reducible* at position p by the rewrite rule $l = r \in \mathcal{R}$ iff there is a substitution σ such that $\sigma(l) = t|_p$ and the reduction is denoted by $t \to t[\sigma(r)]|_p$

a term t is *irreducible* with respect to $\mathcal{R}$ iff no rule of $\mathcal{R}$ can be used to reduce t

t' is a *normal form* for term t if there exists a reduction sequence $t \to t_1 \to t_2 \ldots \to t'$ and t' is irreducible

term t is *narrowable* at non-variable position p ($t|_p \notin V$) if there is a partitioned substitution (σ, θ) which is a *most general unifier* of $t|_p$ with the left-hand side of some rule $l = r \in \mathcal{R}$ (with variables renamed to ensure $Vars(t) \cap Vars(l = r) = \emptyset$) such that $\sigma(l) = \theta(t|_p)$ and the narrowing step can be denoted as $t \rightharpoonup_{[p,l=r,\theta]} \sigma(t[r]|_p)$. Narrowing is thus a proper extension of reduction.

Narrowing provides a sound and complete method which can be used to solve equations with respect to a confluent and terminating set of rules $\mathcal{R}$ [46]. However, the process of narrowing is non-deterministic, with narrowing steps proceeding for each rule whose left-hand side is unifiable with a subterm (redex) of the expression. The application of each rule to each potential redex yields a huge search space with many infinite paths even for simple programs, and it is a fruitful research topic to analyse which restrictions are acceptable to limit this expansion. Most work has been applied to constraining the selection of the redex for the next narrowing step. These refinements include:

- *Basic narrowing.*
  Hullot in [46] describes an optimisation in which redexes are selected for narrowing only if they are part of an original program clause or goal, rather than new terms resulting from previous substitutions. This restriction results in a smaller search space than simple narrowing, but is still sound and complete for a confluent and terminating equational system. A significant advantage of this method is that narrowing positions can be identified at compile time permitting a more efficient implementation.

- *Term ordering.*
  For a certain class of programs, solutions can be computed with the redexes being selected in a *left-to-right* (or other) order. This restriction, summarised by Hanus in [41], results in an incomplete search process, and the implications have not been widely researched.

- *Innermost narrowing.*
  In constructor-based programs, solutions can be found by consistently selecting the innermost term as the redex to be narrowed, and the procedure corresponds to eager evaluation in functional languages. Innermost narrowing is in general incomplete, but has been shown to be complete if all functions are *everywhere defined* (also called totally defined) such that the only irreducible ground terms are constructor terms [38]. Innermost narrowing can be combined with basic narrowing, and *innermost left-to-right basic narrowing* has been shown in [15] to be equivalent to SLD-Resolution if the functional program is transformed to a pure logic program by the process of *flattening*. This transformation requires that each functional term is given a relational representation with a logical variable to contain the result, and nested functional terms are converted to an ordered sequence of these

relational goals with the innermost functional term as the leftmost relational goal.

- *Normalisation and rejection.*
  This technique provides the opportunity of eliminating unnecessary narrowing derivations. In solving an equation $t_1 = t_2$ in the context of an equational system $\mathcal{R}$, the basic approach is to perform a normalisation step on t_1 and t_2 (rewriting them to their normal form) so that the outermost function symbols can be compared before narrowing. If the symbols are for different *constructors*, then the derivation can be terminated at this point.

- *Outermost.*
  This is the converse of innermost narrowing, selecting the outermost defined function term as a candidate for a narrowing step. The procedure is analogous to lazy evaluation in functional languages, but it is incomplete. Outermost narrowing requires the equational system to be terminating (and confluent), a condition violated by the inclusion of infinite data structures, and to address this issue *lazy narrowing* has been investigated [64], in which inner terms are narrowed if their value is *needed* in a later outer narrowing step.

2.2.7 Extended Prolog *call* semantics

Chapter 5 provides a detailed discussion of the capabilities of the function evaluation meta-relations `call/N` and `apply/3` reviewed in this section.

As was noted in section 2.2.1, a function of N arguments can be replaced with a relational procedure of $N+1$ arguments with the additional argument to hold the result.

For comparison, definitions of an integer `plus` function and relation in PrologPF and standard Prolog are:

```
fun plus(X,Y) = X + Y.       % PrologPF function definition

plus(X,Y,Z) :- Z is X + Y.  % standard Prolog clause definition
```

The definitions mean that:

- The appearance of an argument term `plus(3,4)` anywhere in a PrologPF program is equivalent to the use of the constant `7`.

- A goal `plus(3,4,R)` in a standard Prolog clause body will succeed with the variable binding {`R/7`}.

Given the sample definition for the function `plus` it is reasonable to ask for the meaning of the term `plus(3)`. Functional languages such as ML [55, 61] encourage the use of functions with a reduced number of arguments as a mechanism to introduce higher-order programming. This technique of *currying* [33, 67] is central to the design of the higher-order functional aspects of PrologPF. The partial application of a function such as `plus(3)` evaluates to a nameless function which will add `3` to its argument. The technique allows powerful use of higher-order functions such as `map(F,L)` which applies the function `F` to each element of the list `L`. For example `map(plus(3),[1,2,3])` evaluates to `[4,5,6]`. The technique is general, such that `map(plus(3))` represents the function which adds `3` to each element of a list.

The relational representation of the function in standard Prolog does not include any support for the straightforward use of currying. Two proposed library additions to Prolog to support the higher-order use of the `plus(X,Y,Z)` definition given in the example above are `call/N` and `apply/3` [58]. Chapter 5 makes a detailed comparison of these meta-relations with the approach used in PrologPF.

2.2.7.1 `call/N`

In standard Prolog [35], `call(A)` treats `A` as a goal and calls it. For example, `A = plus(3,4,R), call(A)` is equivalent to `plus(3,4,R)`. `call/N` [58] extends the standard `call` meta-relation to more than one argument. `call(A,B`$_1$`,B`$_2$`,...,B`$_n$`)` calls `A` with additional arguments `B`$_1$`...B`$_n$.

The higher-order use of the relational definition of `plus` with `call/N` is illustrated with the following simple example:

```
A = plus(3), call(A,4,R).
```

The technique relies upon the conventional use of the last argument as the result of a functional computation (`R` in the example). The 'flat' style of programming is retained both for the function definition and the higher-order application, such that `call/N` continues the convention of last argument as result.

2.2.7.2 `apply/3`

Naish argues in [58] that an application meta-relation `apply/3` provides more general support for higher-order programming than `call/N`. `apply/3` treats all function applications as to one argument. For example `plus(3,4,R)` is equivalent to `apply(plus,3,Plus3), apply(Plus3,4,R)`.

The semantics of `apply/3` diverge from `call/N` when higher-order intermediate results are returned. For example, the goals `call(plus(1),2,X)` and `apply(plus(1),2,X)` both bind `X` to the number 3. However, whereas `call(plus,2,X)` results in an error or fails, `apply(plus,2,X)` binds `X` to a representation of a function which adds 2 to its argument.

2.3 The Delphi Machine: previous work

A number of researchers have contributed directly to the development of OR-parallel systems based upon the use of oracles and recomputation for execution of pure Prolog programs:

1. Clocksin and Alshawi created the first simulation of the Delphi Machine and proposed a number of strategies for OR-parallel execution of pure Prolog programs [28], summarised in [26].
2. Wrench investigated the extension of the Delphi principle into a system providing both AND-parallelism and OR-parallelism [76].
3. Klein implemented a Prolog compiler targeting a modified Warren Abstract Machine [1] with additional instructions for the creation and interpretation of oracles. Additional strategies were tested with this compiler [49].
4. Barham focussed upon the issue of distributed control of the multiple path-processors, implementing a hierarchical control system [10].
5. Saraswat provided a detailed performance analysis of the existing Delphi implementation, adding new scheduling strategies and a theoretical analysis of the run-time to delivery of the first solution [66].

That work is summarised in the following sections.

2.3.1 The Delphi principle

A brief overview of the Delphi principle for the execution of logic programs was given in Chapter 1.

From [66]: *In an OR-only tree, if each path is executed by exactly one processor then the total execution time required to cover the complete tree has to be the time taken to execute the longest path within the OR-only tree.*

The OR-only tree from Figure 2.3 can be annotated with choice indexes, resulting in Figure 2.6.

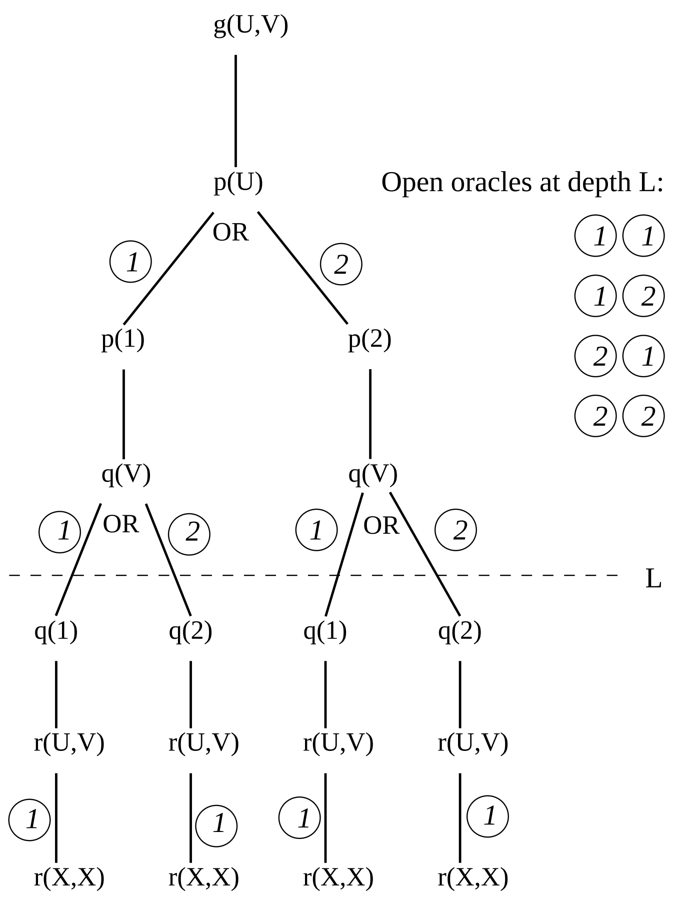


Figure 2.6: Oracles within the OR-only tree from Figure 2.3.

Figure 2.6 shows that at the depth indicated by the dotted line, there are four branches available for further execution. The branches can be labelled *[1,1]*, *[1,2]*,*[2,1]*, and *[2,2]* respectively, each label being the *oracle* uniquely defining that branch in the OR-only tree.

The OR-only tree for the query `g(U,V)` has a maximum of four OR-parallel paths to be allocated to available processors, and the work can be distributed by a control processor sending each a copy of the program and one of the oracles. The path processors can either search the whole subtree below the assigned oracle, reporting back solutions, or they can limit their search in some way, reporting back solutions plus the status of oracles within the subtree. The behaviour of the control processor in allocating work and the associated behaviour of the path processors on receipt of one or more oracles forms the *scheduling strategy.* The simplest strategies for the path processors are to either search fully the subtree below an assigned oracle, or to limit the search to the next choice point in the subtree, reporting back the status of the oracle extensions representing the new branches.

The use of recomputation to reconstruct the environment at a given branch of a tree, with oracles used to define the path from the root to the branch, provides a flexible mechanism for the OR-parallel distribution of work using a variety of strategies. The execution model implemented in PrologPF provides a vehicle for the evaluation of different strategies through the basic support provided for:

1. the accumulation of a current oracle recording the sequence of clauses used as the search progresses,
2. following an assigned oracle, and
3. searching a subtree below an assigned oracle to a specified depth, and reporting any solutions and the status of open oracles at that depth. Different specified depths permit fundamentally different types of strategy:

   zero: simply report back the status of the assigned oracle

   1: extend the search to the next choice point (i.e. increase depth by one)

   integer > 1: perform a partial search within the assigned depth and report back solutions and open oracles.

   -1: search the whole subtree and report solutions or failure

The intermediate case in PrologPF, where the search of a subtree is constrained within a depth parameter, is a special case of a more general solution where the search bound might be specified with a boolean function. That is, PrologPF assumes a search limit function `fun inside_limit() = Depth < Depth_limit` where alternatives might use time or number of choice points traversed.

The use of depth limited search to produce an effective one-time partitioning of the program is described in Chapter 3. Chapter 8 describes an extension to the technique using the interruption of busy processors with recursive application of the partitioning algorithm to repeatedly reallocate work from busy to idle processors.

2.3.2 Architecture

Figure 2.7 illustrates the distributed architecture suited to exploitation of the Delphi principle and targeted by the PrologPF compiler.

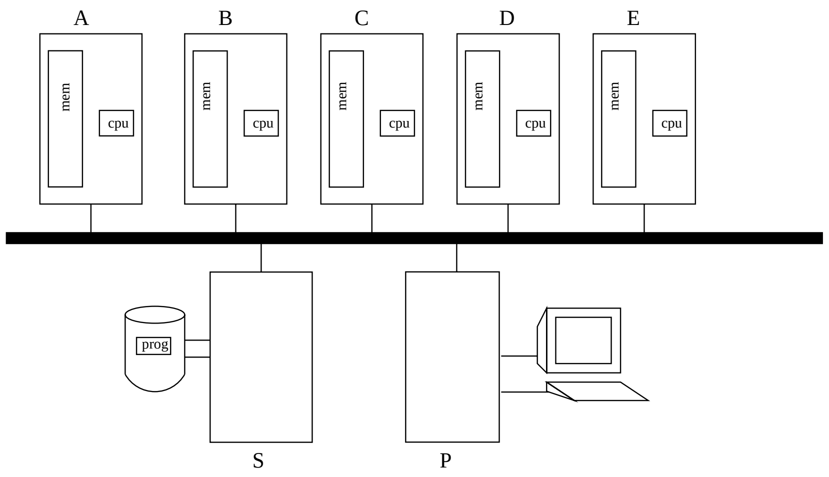


Figure 2.7: Distributed target architecture of the Delphi machine.

The machines labelled **A** through **E** represent path-processors connected to a general purpose peer-to-peer network, represented in the diagram as an Ethernet. Each path processor loads a copy of the program from the file server **S**. Work scheduling instructions are communicated to the path processors from the control processor labelled **P**.

While the diagram shows the interconnection network as an Ethernet, the strategies developed with PrologPF and earlier implementations of the Delphi machine aim to minimise the amount of communication between processors, and a lower performance network could be used. Oracles, in association with the pre-loaded user program, provide an extremely compact representations of the environment at a given point in the search tree. This greatly reduces the amount of data to transferred in the assignment of work, as a trade-off for the recomputation overhead.

Some scheduling strategies such as breadth-first partitioning, described in [66] and Chapter 3, require no communication from the path processors after the initial assignment of work except to return solutions and indicate completion.

The DelphiKS implementation used by Klein and Saraswat in [49, 66] uses an NFS[2] file server for the distribution of the compiled program. The same technique is used by PrologPF. An extended control processor could communicate the compiled program over the same virtual connection used to send oracles and receive results, such that the connection of every path proces-

[2]Network File System

sor to a common file server would be unnecessary. A similar benefit could be gained through the use of FTP[3] rather than NFS. The earlier implementations of the Delphi machine and PrologPF make no use of broadcast techniques for the distribution of the program or other scheduling information, and the initial program load times from the shared file server have not been included in the performance measurements.

In [10], Barham proposes a hierarchical control mechanism for the scheduling of work on the distributed Delphi machine, and PrologPF provides a general hierarchical communications structure (described in Appendix A.4) although its use in the current implementation is limited.

2.3.3 Oracles

The use of oracles assumes a the treatment of the alternative clauses in a procedure as an ordered list, such that the clauses can be numbered 1 to N in their textual sequence in the program.

In PrologPF an *oracle* is a list of integers, each representing the number of the choice point to be selected at each branching point in the OR-only tree.

An oracle can be communicated from the control processor to a path-processor to indicate a subtree for search, or a path-processor can return a set of oracles representing unsearched branches in its allocated subtree. PrologPF can also return the oracle representing the point at which each solution was found.

It has been noted in [28, 49] that each n-ary OR-only tree resulting from the direct transformation of the problem AND-OR search tree has an equivalent binary representation. The transformation from n-ary tree to binary tree requires the nominal insertion of binary nodes above each branch point with more than two branches. The oracles used within this transformed tree are sequences of *bits*, leading to a very compact representation of the environment at any point in the binary tree. The implementation of the Delphi machine in [49] uses a special instruction `setmax` to record the number of alternative clauses N in a given procedure, such that $\log_2 N$ bits will be used from the oracle to define or record the choice selected.

The binary representation of oracles may be the most compact form, optimised for communication across a relatively slow network. The list of integers used in PrologPF is a compromise designed to facilitate debugging and external interpretation of the oracles flowing in the network. However, after the right number of bits is picked off an assigned oracle within the

[3]File Transfer Program

Delphi machine the treatment of the clause number is the same.

An alternative representation of oracles with the same space requirement as the integer list but a more efficient execution would be to record the relative label addresses of each selected clause in the compiled WAM program. The path processor would then follow an oracle by treating it as a sequence of direct jumps. This approach is yet to be tested.

The nature of the breadth-first partitioning (BFP) strategy means that the oracles can be generated *locally*, i.e. within each path processor, during the distributed one-time work assignment phase. This means that *no* oracles are actually communicated across the network. BFP is described below and in detail in Chapter 3 and [66].

2.3.4 Delphi scheduling strategies

A simplistic implementation of the Delphi machine has a fixed number of *path processors* and a *control processor*. The control processor maintains a queue of oracles representing portions potential paths, and sends oracles from this queue to idle path processors [6]. In following an assigned oracle, a path processor can arrive at three possible results:

1. **success:** a solution is found
2. **failure:** the execution path terminates in failure
3. **open:** the assigned oracle leads to a node in the search tree with further branches to be explored

Within this framework, any algorithm can be employed by the control processor to determine the generation of oracles and the assignment of oracles to path processors. Similarly, in the third case, the path processor can continued the search or report the status back to the control processor.

The algorithms used in the control processor for the allocation of work, and in the path processors for the third case listed above, form the scheduling *strategy*. A number of strategies have been tested in the original prototype [6], the subsequent DelphiKS implementation [49] and PrologPF. The results are summarised below.

2.3.4.1 Non-backtracking partitioning strategies

Non-backtracking strategies assume the capability of a path processor to *follow* an oracle and report solutions and status, but do not assume a capability for subsequent backtracking search if open branches are found. The strategies illustrate the flexibility of the Delphi principle, but perform badly for most Prolog programs.

- **Brute force strategies**.
  Strategies of this type do not reduce the search space by accumulating information on previously completed paths. Two examples of brute force strategies proposed by Clocksin and Alshawi in [28] are:

  Random. The control processor generates random oracles for allocation to idle path processors. The path processors report the status of the assigned oracle back to the control processor and return to the idle state. This process is repeated until a solution is found.

  Incremental. The control processor generates all possible oracles in ascending sequence. Oracles of length *one* are allocated first, then all oracles of length *two* and so on. As with the random strategy, the path processors report the status of the assigned oracle and become idle until the assignment of another. The algorithm represents an iterative deepening strategy.

- **Strategies recording incomplete paths.**
  The effectiveness of the control processor in assigning oracles can be greatly improved with the use of a data structure to record those oracles that have been found to lead to open branches. Oracles sent to path processors can be limited to extensions of oracles previously returned. Two strategies of this type evaluated by Klein in [49] differ in the behaviour of the path processor on the assignment of an oracle:

  Expanding a job. On the assignment of an oracle, the path processor follows the oracle and reports *failure* or a solution (*success*) if the oracle leads immediately to either. If the oracle leads to a branching point in the search tree, the oracles representing each branch are returned to the control processor, and the path processor becomes idle. These oracles are added to the queue in the control processor for redistribution to idle path processors.

  Branch by branch. This strategy is similar in principle to the *expanding a job* strategy given above. The strategies differ when the oracle assigned to a path processor leads to an *open* node. With the branch by branch strategy the path processor will report back the oracles representing all the new branches except one, and will continue the search along that path. If the new oracle leads to *success* or *failure*, that will be reported and the path processor will become idle. If the new oracle leads to another branching point (i.e. is *open*), again the oracles representing all the branches at that node except one are returned to the control processor. The path processor then continues with the newly selected oracle.

Partitioning with oracle buffering. This strategy is an extension to *branch by branch* given above, with the use of local buffering in the path processors to record the open oracles as the search along the selected branch progresses [66]. When the path processor reaches the end of its selected branch, a new oracle is picked from the local buffer. If the path processor follows a lengthy branch passing many choice points, such that the number of buffered oracles passes a set threshold, then a quantity of oracles will be transferred to the control processor to free up space in the buffer and provide work for other idle path processors. The path processor becomes idle when it reaches the end of its selected branch and the local buffer is empty.

2.3.4.2 Backtracking partitioning strategies

These strategies assume the capability of the path processor to independently search the subtree beneath an assigned oracle. The strategies differ in the method used to determine the oracle assignment, how to limit the search within the path processor, and whether to reassign the work in a given subtree after the initial assignment of the defining oracle.

- **Automatic partitioning.** Each path processor is given the program, the number of processors in the group G, and the number of that path processor within the group N[4]. The strategy is able to proceed with no further communication from the path processors except to report solutions and completion. Every path processor uses the parameters G and N to select a branch at each OR-node from the root of the search tree, until it has arrived at a unique subtree. The path processor then searches that subtree using Prolog's normal depth-first left-to-right execution strategy. In [49][5], Klein proposes three algorithms for a path processor to select a branch at each OR-node:

Partition right. Each path processors starts at the root of the search tree. At each OR-node with a number of branches denoted M, the $M-1$ path processors with the lowest unique processor numbers each select in order the left-most $M-1$ branches. The remaining $G-(M-1)$ processors follow the last (right-most) branch, to be

[4]Klein in [49] uses U to refer to the unique path processor number

[5]the description of these strategies by Saraswat in [66] differs from that in [49]

similarly distributed at the next OR-node. When an OR-node is reached at which M is equal to the number of path processors remaining, each selects a branch in order of their unique processor number. If an OR-node is reached with M larger than the remaining pool of path processors, then the branches are assigned from the right to the remaining path processors in descending order of unique processor number N, with the path processor with the lowest N taking all the remaining left-most branches. The partitioning of the sample search tree used in [49] with the *partition right* algorithm is shown in Figure 2.8.

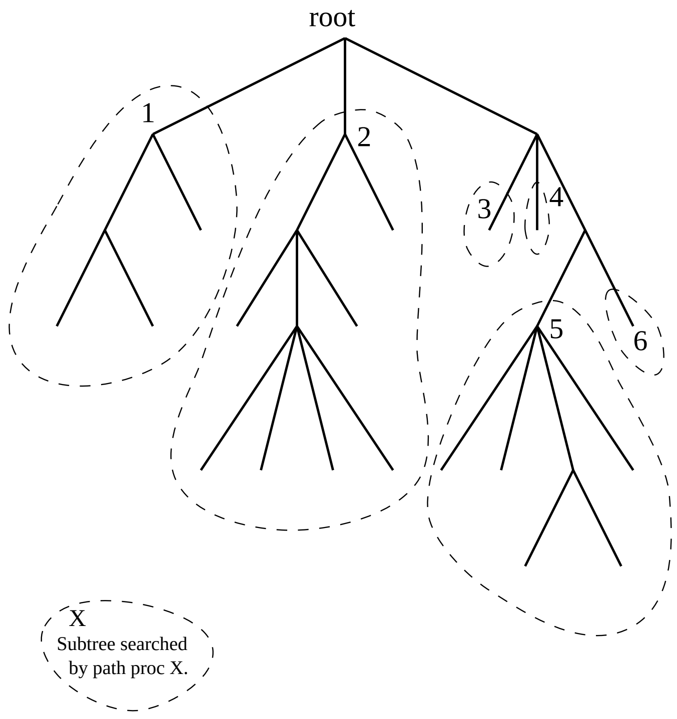


Figure 2.8: *Partition left* of sample tree in [49] for $G = 6$.

Partition left. This strategy is the same as *partition right* given above, except that the $M - 1$ path processors are assigned one to a branch except the left-most, with the remaining $G - (M - 1)$ all taking the left-most branch.

Partition central. This algorithm does not have the extreme left or right assignment bias of the other two automatic partitioning strategies. As with the other two strategies, when the number of branches M equals the number of path processors, then the path processors are assigned one branch each. When the number of available path processors exceeds M, the processors are divided as evenly as possible across the branches. Where the division results in a remainder, one additional processor is assigned to each branch starting from the left. When the number of available processors is less than M, the branches are allocated to processors as evenly as possible. Where the division results in a remainder, one additional branch is assigned to each processor starting with the lowest N. The example tree from [49] with the assignment of subtrees to six path processors is shown in figure 2.9.

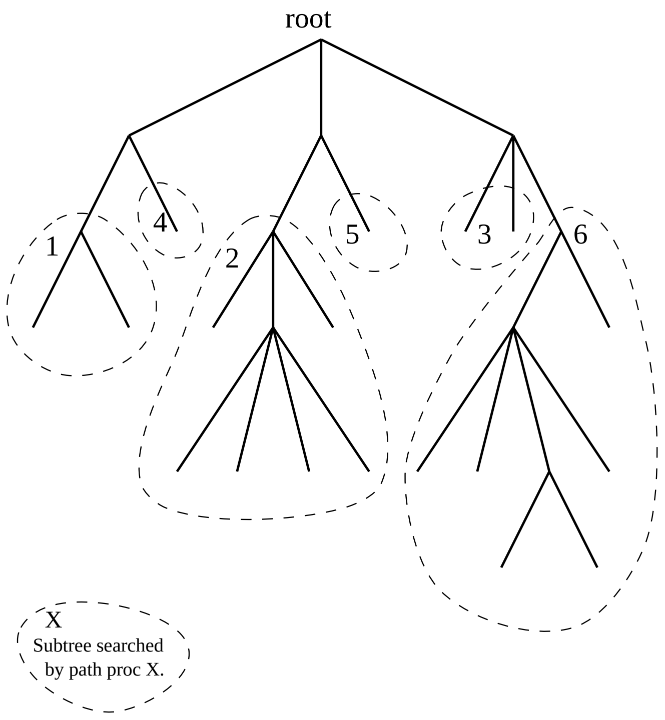


Figure 2.9: *Partition central* of sample tree in [49] for $G = 6$.

- **Reassign-job.** This scheduling strategy was proposed by Klein in [49], with further analysis by Saraswat in [66]. The automatic partitioning strategies perform badly with a search tree with many short branches. Path processors assigned these branches quickly become idle and contribute no further to the OR-parallel search. It is common for a sample program to be reduced to execution on a single path processor within milliseconds of startup [66]. The *reassign-job* mitigates this problem with the following extensions:

1. the control processor maintains a list of idle path processors as each completes its assigned subtree
2. busy path processors pause and report the oracle representing their current position on reaching a defined *check-in interval*
3. the automatic partitioning algorithm is modified to perform splitting only after the path processor has followed an assigned oracle

The check-in interval is a constant, setting a *depth* limit, and the path processor reports its current oracle to the control processor on reaching this limit and becomes idle. On receiving an oracle, the control processor initiates automatic partitioning on all idle processors including the processor which reported the oracle, with the reported oracle defining the new root for the splitting process. Thus the subtree previously allocated wholly to the busy path processor is reassigned to a group of processors.

- **Breadth-first partitioning.** This strategy is suggested by Alshawi and Moran in [6], and was implemented as an extension to DelphiKS by Saraswat [66]. The strategy greatly improves the granularity of task assignment over the earlier strategies, and is used by PrologPF. The *reassign-job* strategy given above seeks to mitigate the problem of early completion and subsequent idleness of many path processors in the *automatic partitioning* strategies by continually redistributing work from busy processors to idle ones. An alternative solution is to more effectively allocate the work in the initial distribution, and the breadth-first partitioning strategy achieves this.

  The scheduling algorithm proceeds in two phases, illustrated in Figures 2.10 and 2.11. In the first phase shown in figure 2.10, the control processor executes the user program to a limited depth in the OR-only tree. In [66] and throughout this document, this depth limit will be called L. Within this depth limit any solutions are reported, and the open oracles of length L recorded in a buffer. It should be noted that the search up to this limit proceeds with the normal depth-first left-to-right strategy of standard Prolog. The open oracles (A...E in figure 2.10) can then be allocated to path processors, for example allocating the nth oracle to path processor n mod G where G is the number of processors available. In the second phase (figure 2.11) each path processor *follows* each allocated oracle and searches the corresponding subtrees using the standard Prolog depth-first left-to-right algorithm, returning solutions and reporting completion. Figure 2.12 compares the possible assignment of path processors to subtrees with the *partition central* algorithm of automatic partitioning versus breadth-first partitioning with a suitable choice of limit L, for a program with many short branches. The tree is typical of the subtrees

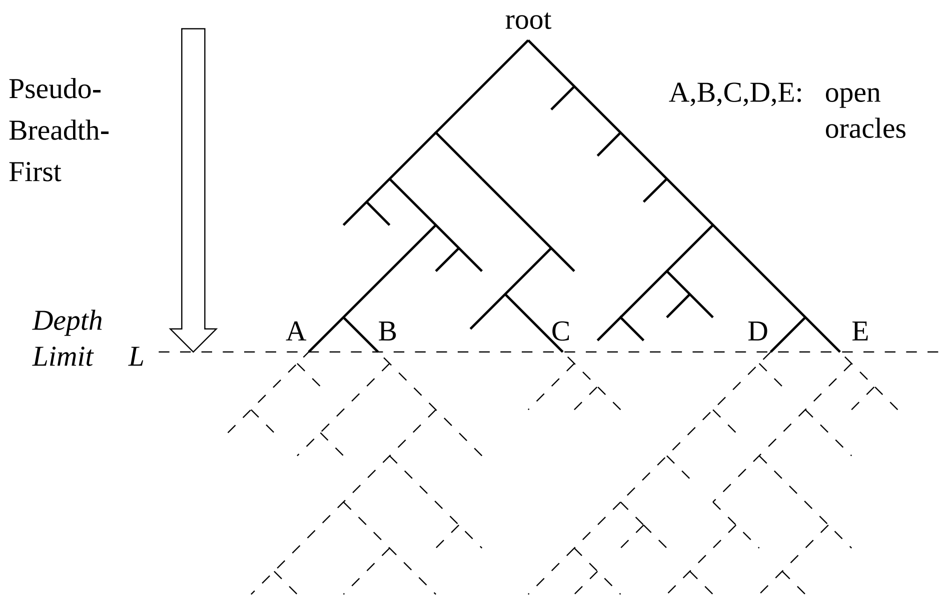


Figure 2.10: First phase of *breadth-first partitioning.*

found in the deterministic execution of relations such as `member` where one rule represents the recursive case, and the other the termination condition. Whether the tree is left- or right-biased depends purely on the ordering of the clauses in the procedure, and the *partition-left* and *partition-right* algorithms in automatic partitioning are critically dependent upon the right match. *BFP* is thus potentially less affected by the systematic presence of many short branches in the search tree, but is dependent upon a good choice for L.

The open oracles at depth L can be generated concurrently by all path processors, and a local algorithm can be used within each path processor to determine a unique subset of the oracles to be searched. With this approach, each path processor needs only a copy of the user program, and the values of G, N, and L (number of processors in the group, unique processor number, and depth limit), for execution to proceed. This is the technique used in [66] on the Delphi machine and in PrologPF.

The BFP algorithm is described in detail in [66] and analysis of the performance of PrologPF for pure Prolog programs using BFP is given in Chapter 3.

- **Partitioning by selective sampling.** The BFP strategy described above improves upon the automatic partitioning strategy by achieving

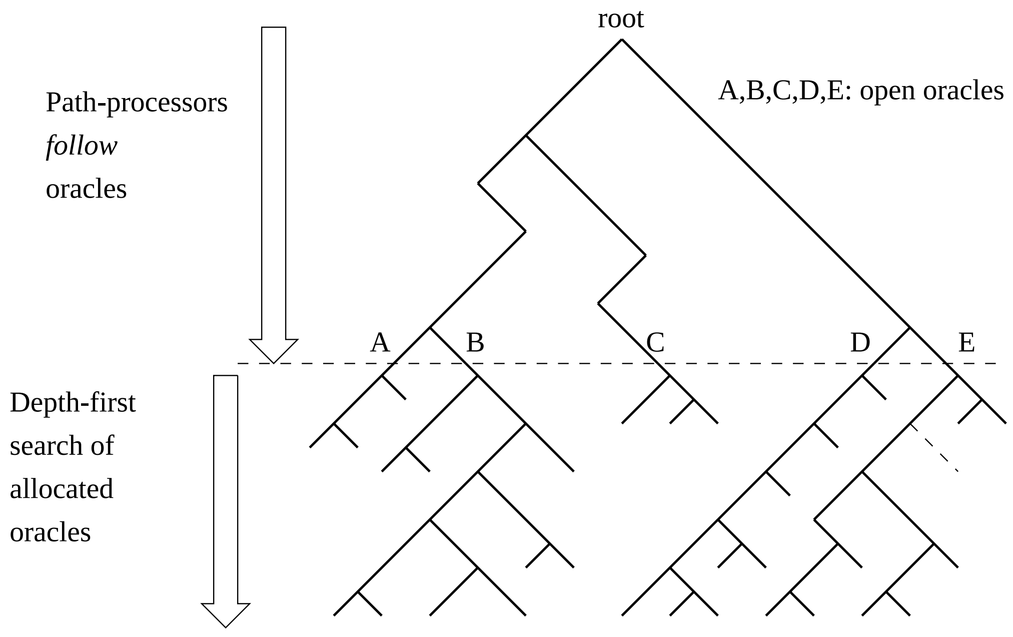


Figure 2.11: Second phase of *breadth-first partitioning*.

a better allocation of oracles to path processors, with less vulnerability to the many short branches in a typical search tree. The depth limit L must generate more oracles than there are path processors for the strategy to be effective. The one-time allocation of oracles requires the cumulative size of the subtrees beneath each assigned subset of oracles to be reasonably well balanced, or the strategy will suffer due to many processors becoming idle at an early stage in the execution. With *partitioning by selective sampling* (also called *PSS*), Saraswat in [66] attempts to improve the effectiveness of the allocation by estimating the size of the subtrees beneath each oracle. The first phase of this strategy is identical to that of BFP. An intermediate phase is added, called the *feedback* phase, in which the subtree beneath each oracle is searched with a limit set on the number of choice points traversed, as a means of estimating the size of the subtree beneath each oracle. A heuristic algorithm [66] is then used to divide the oracles into G subsets with approximately the same cumulative amount of work. Each subset is allocated to a path processor, which proceeds as in the second phase of BFP.

The partitioning by selective sampling strategy did not appear to make an improvement over the underlying BFP. The issues of oracle distribution to available path processors is analysed further in Chapter 3.

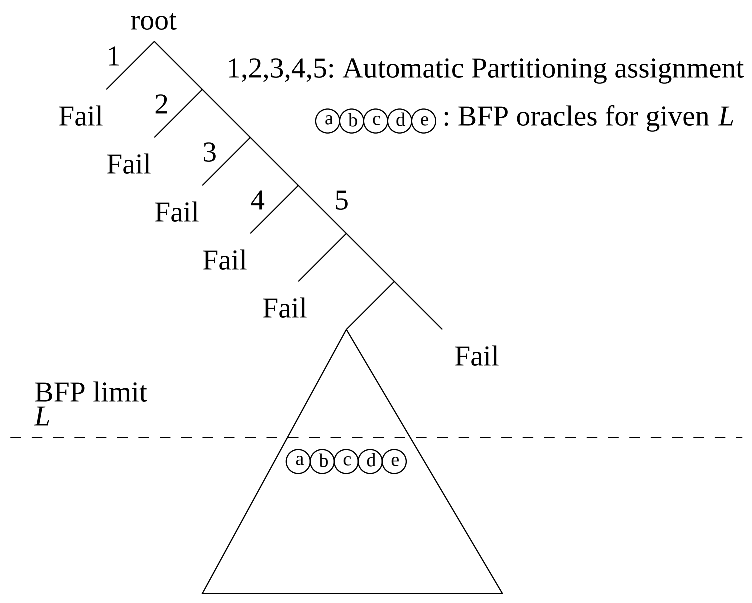

root
1,2,3,4,5: Automatic Partitioning assignment
ⓐⓑⓒⓓⓔ : BFP oracles for given L
1
2
3
4
5
Fail
Fail
Fail
Fail
Fail
Fail
BFP limit L
ⓐⓑⓒⓓⓔ

Figure 2.12: Path processor assignment in *AP* versus *BFP*.

- **Breadth-first partitioning with selective sampling.** This strategy, labelled *BFPSS* by Saraswat in [66], is another extension to breadth-first partitioning to improve the one-time allocation of oracles to available path processors. The first phase proceeds as in BFP and PSS, generating a set of S open oracles at a depth L. As with PSS, the first phase is followed by a feedback phase. BFPSS differs from the previous PSS in the algorithm used to estimate the work in the subtree beneath each oracle.

  In this new strategy, the open oracles found at depth L are divided evenly in consecutive groups among the available path processors. Each path processor treats its assigned set of oracles as an ordered list, and searches fully the subtree below every other oracle, reporting solutions found and recording the amount of work in each subtree. When all the alternate oracles have been fully searched, the path processor calculates estimates for the intermediate oracles as the *arithmetic mean of the sizes of the two adjacent subtrees*, and reports these to the control processor. The control processor sorts all the estimates into descending order, and assigns the associated oracles on a demand basis to the path processors.

  The strategy of *breadth-first partitioning with selective sampling* was found to perform better than the earlier *partitioning with selective* *sampling* but the increased complexity introduced communication and computation overhead such that the performance was less than that of the simpler *breadth-first partitioning* [66].

2.4 Summary

This chapter has presented a summary of other research in the areas of:

- other parallel logic languages,
- functional logic, and
- the development of OR-parallel Prolog on the Delphi machine.

The Delphi principle has been illustrated with examples, and placed in the context of other techniques for OR-parallel execution of logic programs. The alternative approaches have been shown to use environment copying on distributed architectures, and environment sharing on shared-memory multiprocessors. The trade-off of overheads of recomputation versus the communication requirements of environment copying have been highlighted. Current research on functional logic languages have been discussed in terms of their suitability for implementation on the Delphi machine, as an alternative to the use of the extra-logical relation *cut* which is incompatible with the Delphi principles.

The remainder of this dissertation analyses the behaviour of PrologPF in the OR-parallel execution of pure Prolog programs, and reviews in depth the addition of functions to the programming model.

Chapter 3

Prolog with Breadth-First Partitioning

This chapter reviews the performance and behaviour of some test PrologPF programs, with comparison with DelphiKS, the previous implementation of the Delphi machine assessed by Saraswat in [66]. Further analysis is given of the breadth-first partitioning strategy used by PrologPF, particularly of the selection of the depth limit parameter used in the partitioning phase of execution.

3.1 Introduction

The Breadth-First Partitioning scheduling strategy is described in detail by Saraswat in his work on DelphiKS [66] and summarised in Chapter 2 section 2.3.

The execution of the strategy requires that three parameters are passed to the path processor with the associated compiled program:

1. G: the total number of path processors working in parallel
2. N: the unique processor number assigned to a given path processor
3. L: the depth bound at which open oracles are recorded in the initial search phase, for subsequent allocation for execution

In [66], Saraswat analysed a number of benchmarks with several different strategies with processor group sizes (G) from 1 to 30 to assess the efficiency of the speedup of the Delphi machine. This comparative assessment of PrologPF differs from the approach in [66]:

- While [66] evaluates a variety of strategies, this assessment is limited to that found to have the best performance across the range of test programs, namely Breadth-First Partitioning.
- Saraswat included a number of benchmarks known to contain little or no OR-parallelism. Evaluation of these benchmarks with DelphiKS confirmed the expected absence of parallel speedup.
- This study provides a detailed analysis of the behaviour of the strategy with differing values of L, rather than the optimal values of L used for the runtime versus G graphs in [66].

3.2 Benchmarks used

The source code for the benchmarks is given in Appendix B. The appendix also includes graphical representations of the search tree for each benchmark, with the tree for the 8-queens problem repeated here.

3.2.1 8-queens

The 8-queens benchmark finds solutions to the problem of placing eight queens on an 8-by-8 chessboard with no two pieces on the same horizontal, vertical or diagonal line of squares. The simple benchmark finds all 92 solutions, including those which are rotated or reflected versions of other solutions.

The problem has a maximum search depth of 60, and a geometric representation of the complete search tree is given in Figure 3.1. The root of the search tree appears at the top of the figure, and the "depth" value of the ordinate represents the nested depth of subgoals into the search. The OR-only tree contains a maximum of approxiamtely 1500 open branches at a depth of 45.

The other benchmarks have much larger search trees than the 8 queens problem. The complete mapping of the search space resulting in the tree diagram in Figure 3.1 has not been repeated here for the other benchmarks, but are in Appendix B.

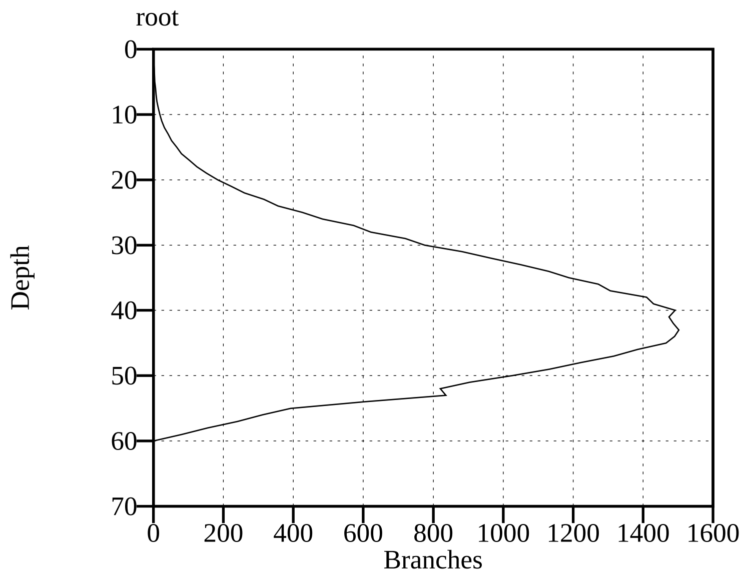


Figure 3.1: Search tree for 8 queens problem.

3.2.2 10-queens

This benchmark is identical to the 8-queens benchmark described above, except the problem is extended to place ten queens on a 10-by-10 chessboard.

The problem has 724 solutions, and the maximum depth of the search tree is 85. The OR-only tree has a maximum of approximately 26000 branches at a depth of 65.

3.2.3 Pentominoes

A pentomino is a shape made up of five equal-sized squares, for example in the shape of a letter `T` or a letter `L`. There are twelve possible shapes from five connected squares, and the benchmark program `pentbook` attempts to fit these pieces on a 20-by-3 board. Each piece has a number of possible orientations and reflections, and can be used only once. The benchmark finds 8 solutions. The maximum depth of the search tree is 90, with the maximum of approximately 100,000 branches open at a depth of 58. The Prolog source for the benchmark `pentbook` is contained in Appendix B.2.

3.2.4 Other benchmarks in DelphiKS

In addition to the benchmarks discussed above, Saraswat in [66] analysed a number of additional programs which have not been included in this comparison:

FFT: a fast-Fourier transform algorithm which contains no OR-parallelism, and thus showed no speedup on the Delphi machine. Similarly, no speedup would be expected with the use of PrologPF.

Adder, Permutation, Balanced: these benchmarks have low runtimes even with a single cpu, and some generate many solutions. For example, the `perm` benchmark compiled with the PrologPF compiler completes in 540 milliseconds on a single cpu, generating 720 solutions. The short runtime and large number of solutions cause the input/output requirements to dominate the execution times in any attempt at parallel speedup. DelphiKS took 7.87 seconds to execute the `perm` benchmark on the same single processor, with more scope for parallel improvement. The `adder` and `balanced` benchmarks took 12.61 seconds and 6.23 seconds to execute on a single cpu DelphiKS system respectively.

3.3 Implementation differences

3.3.1 DelphiKS

The previous implementation of the Delphi machine, DelphiKS [49, 66] included support for additional WAM [73] instructions:

onumtry, onumretry, onumtrust: these DelphiKS instructions are analogous to their WAM counterparts [1], building and removing choice points at the entry-points of compiled alternative clauses in nondeterministic procedures. The DelphiKS instructions accumulate the current oracle as the search proceeds, and provide support for following oracles when executing in that mode.

onumsing: this is a special instruction placed at the entry point of the clause where a procedure contains only a single clause, such that no choice point is built and the current oracle need not be extended. The instruction is used instead of the usual nondeterministic instructions listed above when non-backtracking strategies are used.

setmax: this instruction is inserted at the beginning of each procedure, before the first clause. It provides a point at which the number of alternative clauses in the procedure can be recorded. This information can be used to determine the right number of bits to remove from the binary oracle being followed. The `setmax` instruction provides the point at which scheduling decisions can be taken in the path processor. For example, as the number of alternatives is known at this point (in the argument to `setmax`) the incremental extensions to the current oracle can be built and returned to the control processor. Alternatively the current depth can be checked against the depth bound in the breadth-first partitioning strategy.

In addition to the extra instructions, the extended WAM maintains a few data structures, most notably the oracle stack.

3.3.2 Prototype PrologPF with Prolog oracles

In this implementation of PrologPF the user program is transformed to a version in which every relation has additional arguments, such that the current oracle is propagated through the procedure calls. The Prolog support for difference lists and logical variables is exploited for an efficient implementation.

A utility procedure `o_next(N,A)` is provided which returns the next choice index of the current oracle in `N` and accepts the current oracle `A` as an argument. If the program is being used to **build** an oracle then `N` will be returned as a logical variable embedded in `A`, such that the immediately following call of a relation will instantiate that variable to the clause index. For example, the following program illustrates the transformation:

```
a(X) :- b(X), c(X).

b(a).
b(c).

c(c).
```

this is transformed to:

```
a(1,A,[1|E],En,X) :- o_next(N1,A), b(N1,A,E,En,X),
                     o_next(N2,A), c(N2,A,E,En,X).

b(1,A,[1|E],E,a).
b(2,A,[2|E],E,c).

c(1,A,[1|E],E,c).
```

A query such as `a(X)` is replaced with `o_next(N,A),a(N,A,A,X)` in which `o_next(N,A)` provides the first oracle element `N` to be used as the first argument of the goal, and `A,A` represents the empty difference list of the initial oracle.

An oracle to be followed is stored as a Prolog list in a global variable.

The general format of each relation is:

```
       relation_name(cn, orc, orc_hole, new_hole, args...)
```

where:

`cn`: clause number, in textual sequence of procedure.

`orc`: oracle accumulated so far and passed to procedure as difference list.

`orc_hole`: logical variable representing end of `orc`.

`new_hole`: variable returned by procedure as new hole at end of extended `orc` when the procedure succeeds.

The semantics of `o_next(N,A)` are:
on first call:

```
 o_next(N,A)   :-   if (current mode = FOLLOWING an oracle)
                    then N := next oracle element
                    else /* currently BUILDING an oracle */
                        if (current depth = depth limit L)
                        then
                            push A onto oracle stack;
                            fail
                        else
                            increment current depth.
```

and on backtracking:

```
 o_next(_,_)   :-   decrement current depth,
                    fail.
```

At any point during the execution of the program, the logical variable `A` contains the current oracle. The implementation of oracle support using Prolog procedures and data structures provides a flexible tool for the analysis of different strategies, but generates considerable overhead.

3.3.3 PrologPF with C oracles

To reduce the overhead of the use of Prolog data structures to represent oracles and the definition of the utility relations such as `o_next` as Prolog procedures, the oracle support in PrologPF was re-implemented in C.

For simplicity of implementation and debugging, a similar program transformation technique was used, but with more efficient primitives. As an example of the more efficient transformation, the same example will be used as above.

```
a(X) :- b(X), c(X).

b(a).
b(c).

c(c).
```

The user program can be transformed into two subprograms, one suitable for building an oracle as the search progresses, and the other suitable for following an assigned oracle.

As the execution progresses, to accumulate an oracle as the search tree is traversed the sample program can be transformed by the compiler to:

```
% BUILD code

a(X) :- o_build(1), b(X), c(X).

b(a) :- o_build(1).
b(c) :- o_build(2).

c(c) :- o_build(1).
```

The special relation `o_build` is defined as follows:

`o_build(X) :-` *append index X to end of current oracle.*
`o_build(_) :-` *pop last index from oracle,* `fail.`

The `o_build` relation is defined using C macros, and has the side-effect of updating the state of the current oracle. At any point in the search, the current oracle represents the sequence of clause indexes to traverse the tree directly to the current node.

A query such as `:- a(X)` will initially create an oracle *[1]*, and then solve the subgoal `b(X)`, appending *1* to the current oracle and returning with the solution {`X/a`}. Then the subgoal `c(a)` is called, which fails, and on backtracking the `b(X)` goal removes the appended *1* from the oracle and searches for an alternative solution for `b(X)`. The `b(c)` clause is tried and succeeds, appending *2* to the current oracle and returning the solution {`X/c`}. The subgoal `c(c)` is now called, appending *1* to the current oracle and succeeding. The `a(X)` goal succeeds, at which point the current oracle is *[1,2,1]*.

For the purposes of strategies such as breadth-first partitioning, it is useful to accumulate additional state information as the search progresses, particularly the current depth of the search, or equally the length of the current oracle. Breadth-first partitioning requires the use of a depth limit (referred to as L). To constrain the search to within this depth, the `o_build` relation can be made to *fail* whenever this depth is reached. So the extended definition of `o_build` is as follows, with the depth initially zero:

```
o_build(X)   :-  increment depth
                 append index X to end of current oracle.
                 if depth = L
                 then record current oracle and fail.
o_build(_)   :-  decrement depth
                 pop last index from oracle
                 fail.
```

The pseudo-code for `o_build` show it has the following characteristics:

- the first clause succeeds if the current depth is not equal to L
- the second clause always fails
- if the current depth is less than L, the relation always has one solution
- if the current depth is equal to L, the relation always fails
- the current depth can never become greater than L
- when the query completes the depth will be zero, as each increment operation is always followed by an associated decrement.

In the implementation in PrologPF, the current oracle is accumulated in a C array with the current depth as the index.

It should be noted that the *build* code described above records the oracles at points in the search where the current depth equals the depth limit L. In order to *follow* these open oracles, a different transformation of the sample program can be used:

```
% FOLLOW code

a(1,X) :- o_follow(N1), b(N1,X), o_follow(N2), c(N2,X).

b(1,a).
b(2,c).

c(1,c).
```

A query such as `a(X)` will similarly be transformed to `o_follow(N), a(N,X)`. Given an oracle built by the transformed program using `o_build` described earlier, the definition of the special relation `o_follow` is as follows:

`o_follow(X)` :- *return with X = next index from current oracle.*

This definition of `o_follow` is satisfactory when the oracle being followed leads directly to a solution (or failure). If the assigned oracle is in fact *open*, i.e. leads to an intermediate node in the search tree, then additional support is required. After following an open oracle to its end, PrologPF will then continue the search in `build` mode. This can be viewed as generating extensions to the supplied current oracle. This cross-over from the transformed program providing oracle following support to the code to build the oracle extension can be achieved by extending the program transformation with additional clauses:

```
% FOLLOW code

a(1,X) :- o_follow(N1), b(N1,X), o_follow(N2), c(N2,X).
a(0,X) :- a(X).  % crossover clause

b(1,a).
b(2,c).
b(0,X) :- b(X).  % crossover clause

c(1,c).
c(0,X) :- c(X).  % crossover clause
```

The `o_follow` relation is similarly extended to return *0* as an index when it reaches the end of the currently followed oracle. From this point on, all subgoals will map to their *build* counterparts and the search can continue, again using a depth limit if required.

With the breadth-first partitioning strategy the unconstrained search in the second phase means the overhead of the accumulation of the current oracle in that phase is unnecessary. The *crossover* clause could equally transfer execution to the subprogram with no oracle support, for an execution efficiency identical to that of the standard Prolog compiler. However, more complex strategies, particularly those requiring redistribution of work during the second phase, would require the continued accumulation of the current oracle.

As with the `o_build` relation, the `o_follow` relation maintains a global value to represent the current depth in the search. This depth is also the index into the C array containing the current oracle.

In the *follow* transformation described above, the oracle index has been shown as an new first parameter added to the head of each clause. The indexing of the first argument used in most Prologs and the C macro implementation of `o_follow` means the following of oracles can be particularly efficient.

The transformations of the sample program into the *build* code and *follow* code results in a program suitable for executing many scheduling strategies with small adjustments to the definitions of `o_build` and `o_follow`. The transformations described above are applicable to all implementations of Prolog providing linkage to C routines or macros.

In PrologPF, a small trade-off has been made against execution efficiency for a simpler combined transformation, retaining the use of `o_build` and `o_follow`. The transformed program in PrologPF is considered to execute in one of two modes, *build* or *follow*, such that the `o_build` and `o_follow` relations behave as before in the *build* and *follow* modes respectively, otherwise they do nothing. The combined transformation used in PrologPF is as follows:

```
a(X,1) :- o_build(1), o_follow(N1), b(X,N1), o_follow(N2), c(X,N2).

b(a,1) :- o_build(1).
b(c,2) :- o_build(2).

c(c,1) :- o_build(1).
```

An associated query such as `a(X)` is similarly transformed to:

```
o_follow(N),A(X,N).
```

The added clause index is moved to the end of the parameter list in the head of each clause, so that Prolog indexing on the first parameter can still be exploited in *build* mode. The breadth-first partitioning strategy is designed to involve much more search (in build mode) than the initial following of assigned oracles by path processors, so the indexing of the added arguments in the implementation described previously has been forfeited.

3.3.4 BFP and oracle allocation algorithm

The implementation of the breadth-first partitioning strategy is the same for the oracle support in C or Prolog:

1. The arguments G, N and L are read from the command line.
2. The depth limit is set to L and a *build* version of the query is called. This results in a stack of oracles in the global array.

3. Beginning with the Nth oracle, and subsequently with every Gth oracle, the path processor executes a *follow* version of the query with the depth limit $L = \infty$. I.e. path processor N will be allocated the ith oracle iff (i mod G) $= N$. Solutions found are reported to the control processor.

4. Completion is reported to the control processor.

3.4 Single CPU performance

The Prolog programs compiled for the Delphi Machine and with the PrologPF compiler can be successfully run on a system with a single cpu simply by specifying the group size $G = 1$, and allocating the selected processor the unique processor number 0. The execution performance of the Delphi implementations can then be compared with each other and with some sequential implementations of Prolog to assess the overhead caused by the use of oracles.

3.4.1 Single CPU comparison: DelphiKS, PrologPF, sequential Prolog

The bar charts in figures 3.2, 3.3 and 3.4 compare the performance of the following systems using a single processor[1]:

DelphiKS: the figures for the single cpu performance of the previous implementation of the Delphi machine by Klein [49] are taken from Saraswat's analysis [66].

PrologPF: these figures refer to the execution of the compiled benchmarks using the prototype implementation of PrologPF using Prolog data structures to hold the oracles, and Prolog procedures to build and follow oracles.

PrologPF(C): these figures are from the benchmarks compiled with the runtime oracle support in PrologPF implemented as C macros (see section 3.3.3).

wamcc: the benchmarks were compiled with the sequential Prolog compiler `wamcc` on which PrologPF is based, and runtimes from a single processor were recorded.

[1]MIPS-based DECStation 3100

Sicstus: the SICStus sequential Prolog compiler (SICStus Version 3 [16]) was used to produce a *compactcode* compiled file, for loading and executing with the SICStus runtime system. The compiler option producing the fastest binaries, *fastcode*, was not available for the MIPS systems used in the test.

As the scheduling strategy is of no purpose in a single cpu environment, the overhead of the breadth-first partitioning strategy was minimised by setting the depth limit $L = 1$. The compiled files produced by PrologPF thus spent very little time in the initial partitioning phase (less than 1ms), and the overhead of PrologPF versus wamcc is caused by the runtime management of the current oracle during the progress of the search.

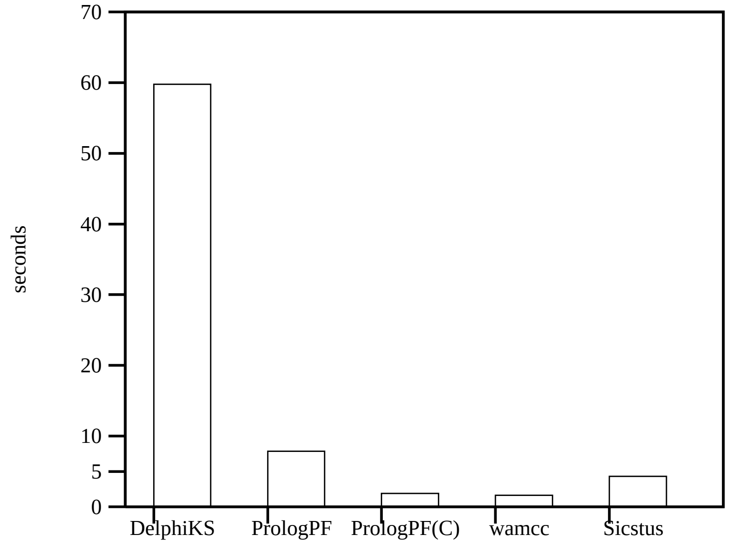


Figure 3.2: Single cpu runtimes for 8-queens benchmark.

The single processor execution times for each benchmark with each compiler are given in Table 3.1.

Benchmark	**DelphiKS**	**PrologPF(Prolog)**	**PrologPF(C)**	**wamcc**	**SICStus**
8-queens	59780	7859	1898	1636	3480
10-queens	1198770	197956	46497	38978	91490
Pentominoes	2824670	1041660	445959	410908	340885

Table 3.1: Single cpu execution times (in milliseconds)

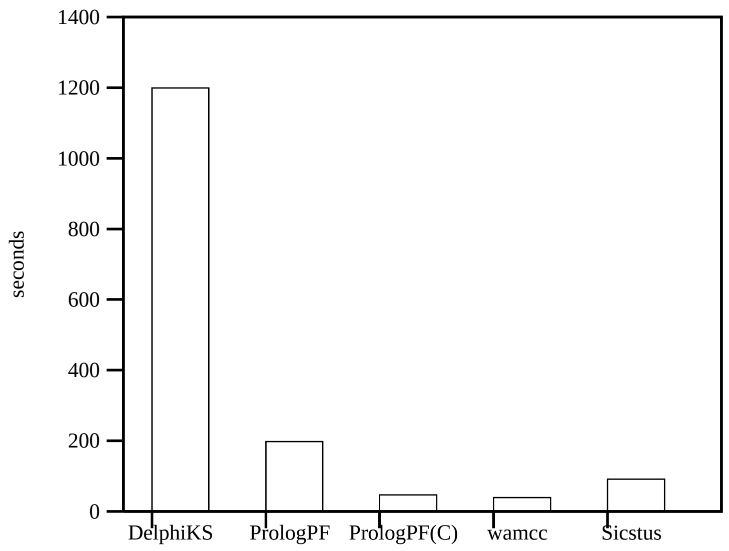


Figure 3.3: Single cpu runtimes for 10-queens benchmark.

The overhead of the oracle support coded with C macros in PrologPF(C) for each of the three sample benchmarks is given in Table 3.2.

Benchmark	**PrologPF(C) overhead**
8-queens	16%
10-queens	19%
Pentominoes	9%

Table 3.2: Overhead of PrologPF(C) oracle support.

3.5 Parallel execution performance

The runtime and speedup figures for the parallel execution of each benchmark are summarised in Table 3.3.

3.5.1 8-queens

The graph showing the reduction in runtime for increasing number of path processors with a suitable partitioning depth is given in Figure 3.5.

With the selected depth limit $L = 21$ for the breadth-first partitioning, the program shows consistently reducing runtimes as the number of path-

Test Program	Total Execution Time (with increasing number of path-processors)											Re-comp. Time	Answers	Partitioning depth	Recorded oracles
	1	3	6	9	12	15	18	21	24	27	30				
8-Queens	1.90	0.753	0.461	0.297	0.258	0.215	0.207	0.176	0.160	0.156	0.141	0.030	92	21	184
10-Queens	46.50	15.88	8.925	5.886	5.070	3.863	3.379	3.485	3.121	2.472	2.519	0.168	724	27	864
Pentominoes	446.0	185.0	122.7	83.02	93.22	71.27	44.69	40.83	51.39	37.26	41.49	0.672	8	21	848
	Speedup factors (with increasing number of path-processors)														
8-Queens	1	2.52	4.12	6.39	7.35	8.83	9.17	10.78	11.86	12.17	13.46	0.030	92	21	184
10-Queens	1	2.93	5.21	7.90	9.17	12.04	13.76	13.34	14.90	18.81	18.45	0.168	724	27	864
Pentominoes	1	2.41	3.63	5.37	4.78	6.26	9.98	10.92	8.68	11.97	10.75	0.672	8	21	848

Table 3.3: Parallel execution performance of PrologPF for appropriate partitioning-depths.

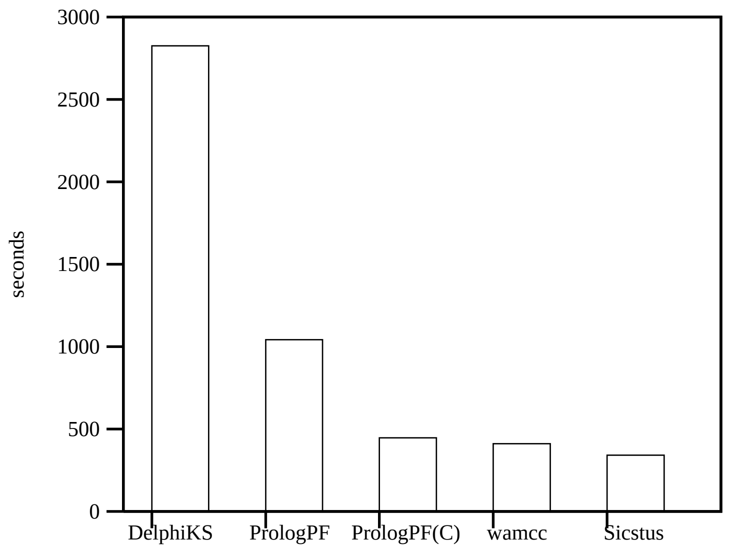


Figure 3.4: Single cpu performance for Pentominoes benchmark.

processors increases from 1 to 30. The 8-queens problem has a well balanced search tree, and is therefore suited to the one-time breadth-first partitioning OR-parallel execution technique used in PrologPF. The graph of runtime ratios representing speedups over the single-cpu case are given in Figure 3.6.

As with the runtime graph, the speedup graph shows that with groups of path-processors in the range $G = 1 \ldots 30$ there is an improvement in performance as processors are added. As will be seen with later benchmarks, the speedup curve illustrates any reduction in efficiency as processors are added more clearly than the equivalent runtime graph. While the speedup curve for the 8-queens benchmark is monotonically increasing, it does not increase directly with the number of available processors G. For example the speedup for $G = 24$ is approximately 12, so 50% of the available processing resource has been effectively applied to the problem.

There are several factors which limit the efficiency of the parallel execution of the problem, and these are discussed in detail in section 3.6.

3.5.2 10-queens

The graph showing the reduction in runtime for increasing number of path processors is given in Figure 3.7; the reciprocal data representing speedups

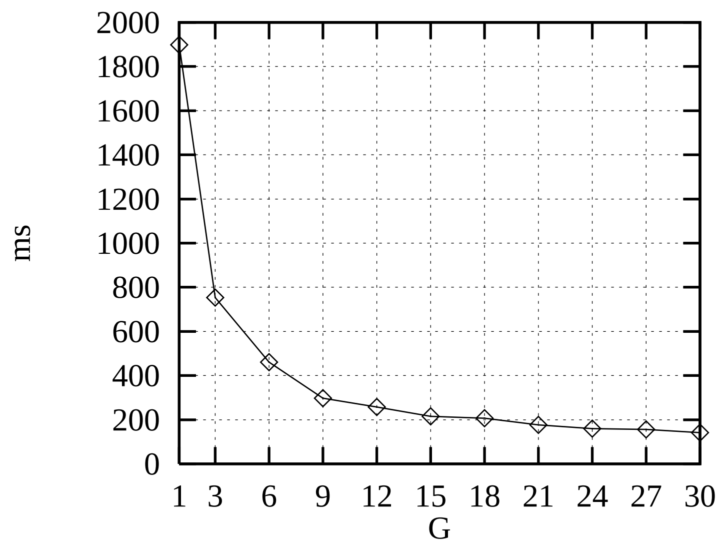


Figure 3.5: Runtimes for 8-queens benchmark for $G = 1 \ldots 30$ and $L = 21$.

over the single-cpu case is given in the graph of Figure 3.8. As with the 8-queens benchmark, a suitable value for the depth limit L has been selected for this initial performance analysis. The issue of the selection of the depth limit is discussed further in section 3.6.

The speedup graph in Figure 3.8 shows a maximum speedup of approximately 19 for any number of path-processors from $G = 1 \ldots 30$. For two values of G, $G = 21$ and $G = 30$ the speedup is actually less than can be achieved with fewer processors. The depth limit L is fixed at $L = 27$ for the parallel execution with each value of G, such that the number of oracles remains fixed at $S = 864$. The variation in speedup arises from the different allocation of oracles to path-processors with the simple modular algorithm used in PrologPF. This issue is discussed further in section 3.6.

3.5.3 Pentominoes

The graph showing the reduction in runtime for increasing number of path processors is given in 3.9, and the reciprocal data representing speedups over the single-cpu case in Figure 3.10.

While showing improvements in runtimes up to approximately 18 path-processors, further increase in G does not result in a reduction of the runtime below approximately 40 seconds. At the depth limit $L = 21$ used in the test

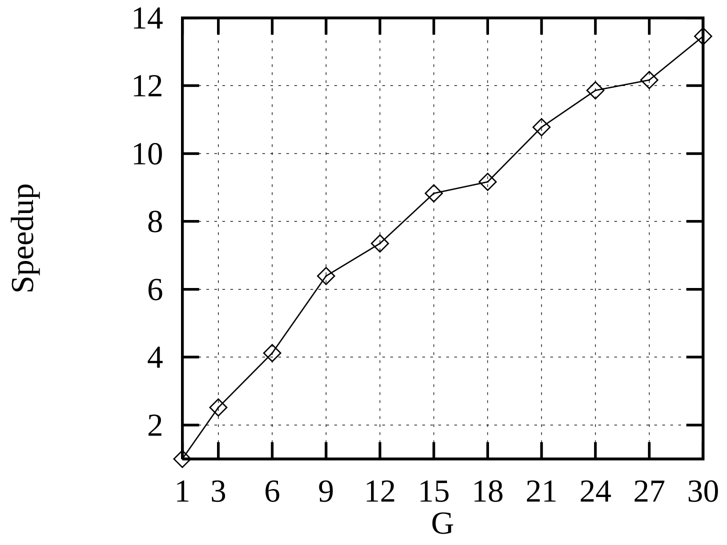


Figure 3.6: Speedup for 8-queens benchmark for $G = 1 \ldots 30$ and $L = 21$.

there are 848 oracles, and one of those oracles has sufficient work beneath it to cause the associated path processor to determine the overall runtime figure. This issue arises because:

- The breadth-first partitioning strategy used in PrologPF performs a one-time allocation of oracles without subsequent work-splitting. The introduction of work-splitting with PrologPF is described in Chapter 8.

- The overall runtime of the parallel execution of the problem is determined by whichever path processor takes the longest time to search the subtrees of its allocated oracles.

The issue of the presence of large outlying oracles is discussed further in section 3.6.

The speedup graph in Figure 3.10 emphasises the limit of reduction in runtime as a maximum speedup for $L = 21$ of 12. In the same graph, the values of G for which the speedup is actually *lower* than for a run with fewer path-processors can be seen ($G = 12, 24$ and 30). As with the 10-queens benchmark, these anomalies arise from the allocation of the 848 oracles at $L = 21$ modulo G to the path-processors (see section 3.3.4).

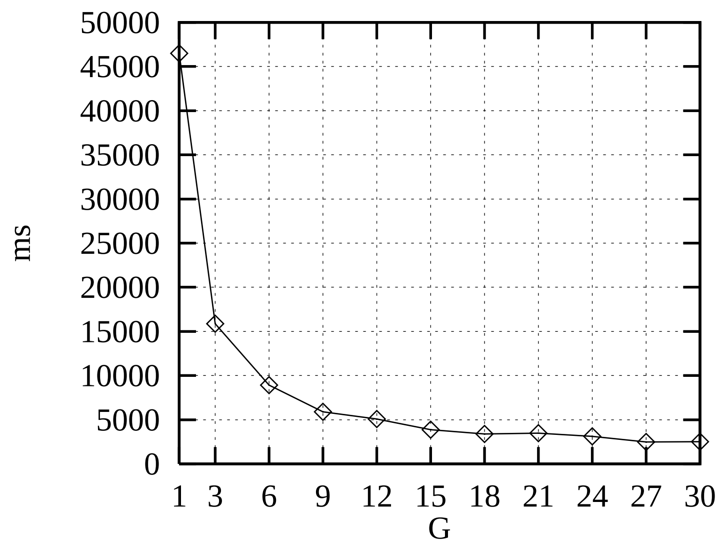


Figure 3.7: Runtimes for 10-queens benchmark for $G = 1 \ldots 30$ and $L = 27$.

3.5.4 Summary

For the benchmarks containing available OR-parallelism tested, PrologPF provides effective speedup over the single-cpu case. The parallel computing environment consists of commonly available workstations interconnected in a typical Ethernet-based intranet. With the one-time partitioning of BFP, no communication is required between the cooperating processors after the initial assignment of the work except to return solutions and report completion.

The parallel speedup provided by PrologPF is limited by a number of factors, including:

- The size of the problem
- The modular allocation of the open oracles
- The one-time allocation of the open oracles, without subsequent work-splitting

These issues are discussed in detail in the remainder of this chapter.

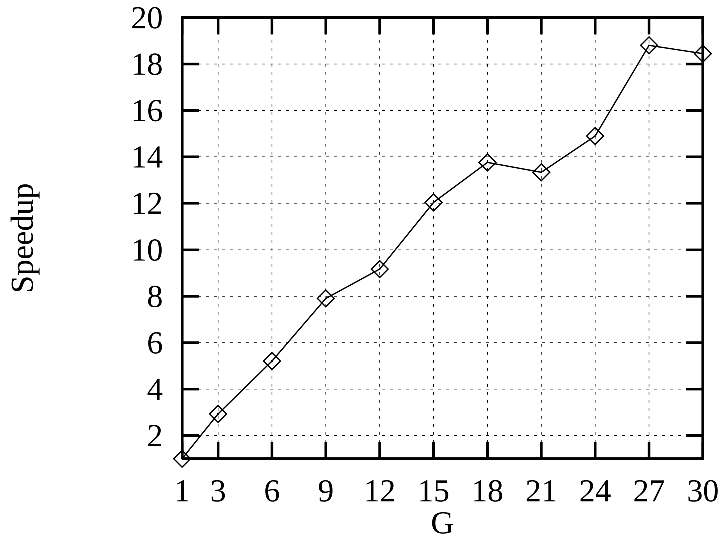


Figure 3.8: Speedup for 10-queens benchmark for $G = 1 \dots 30$ and $L = 27$.

3.6 Issues

The performance tests from the three benchmark programs highlighted three issues affecting the maximum speedup available with PrologPF:

- Although the problems with available OR-parallelism show a speedup with increasing numbers of path-processors, the slope of the speedup graph is less than 1. An ideal parallel implementation with G processors would achieve a speedup of G, and PrologPF falls short of this goal, sometimes dramatically.
- For some values of G the speedup with a given depth limit L is *less* than the speedup with fewer path-processors.
- At some value of G for a given depth limit L, the runtime of the parallel execution of the problem reaches a lower limit, after which no improvement in runtime is available with increasing G.

These issues are apparent even with an optimally selected value of L. The causes are as follows:

1. The initial phase of the breadth-first partitioning scheduling strategy, producing open oracles at the depth limit L, is performed sequentially

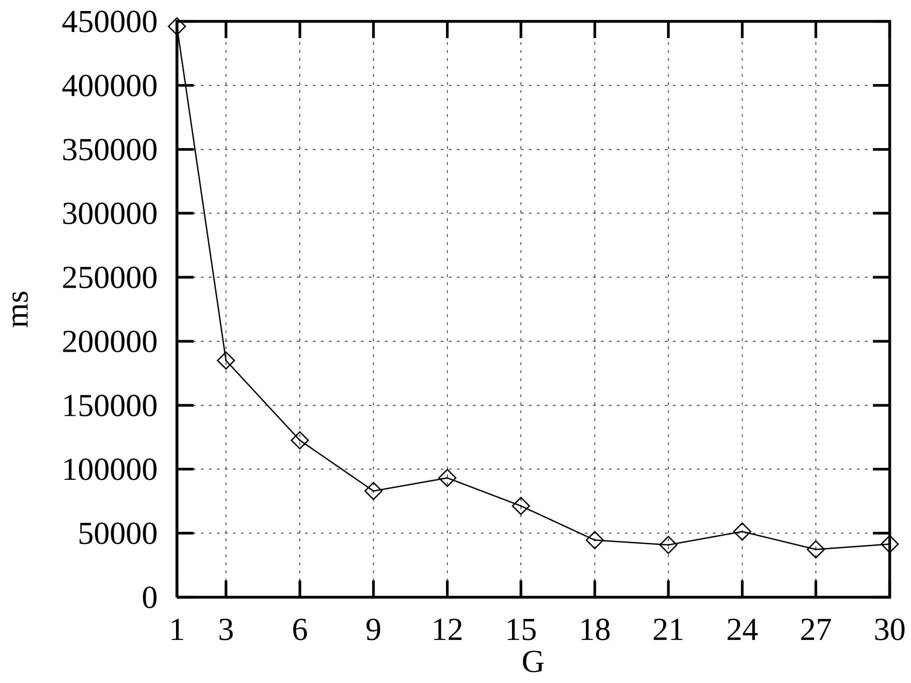


Figure 3.9: Runtimes for Pentominoes benchmark for $G = 1 \ldots 30$ and $L = 21$.

before the subsequent allocation of the discovered oracles to path processors. The time taken for this initial phase places an upper bound on the speedup possible. This problem is most significant for problems with relatively small search trees.

2. PrologPF uses a simple modular allocation algorithm, given in section 3.3.4, to allocate the discovered oracles to the G available path processors. No estimate is made of the size of the subtree below each oracle before it is assigned to a path processor. Some values of G may cause the distribution of the oracles to be particularly unfavourable. As the parallel runtime with one-time partitioning is dependent on the maximum runtime of any contributing path processor, a bad distribution of oracles can result in a runtime worse than that of a more favourable distribution on fewer path processors.

3. With a selected value of the depth limit L, a fixed number of open oracles is produced. As PrologPF with one-time partitioning performs no work splitting after the initial assignment of an oracle, the parallel speedup is limited by the runtime of the search of the largest subtree.

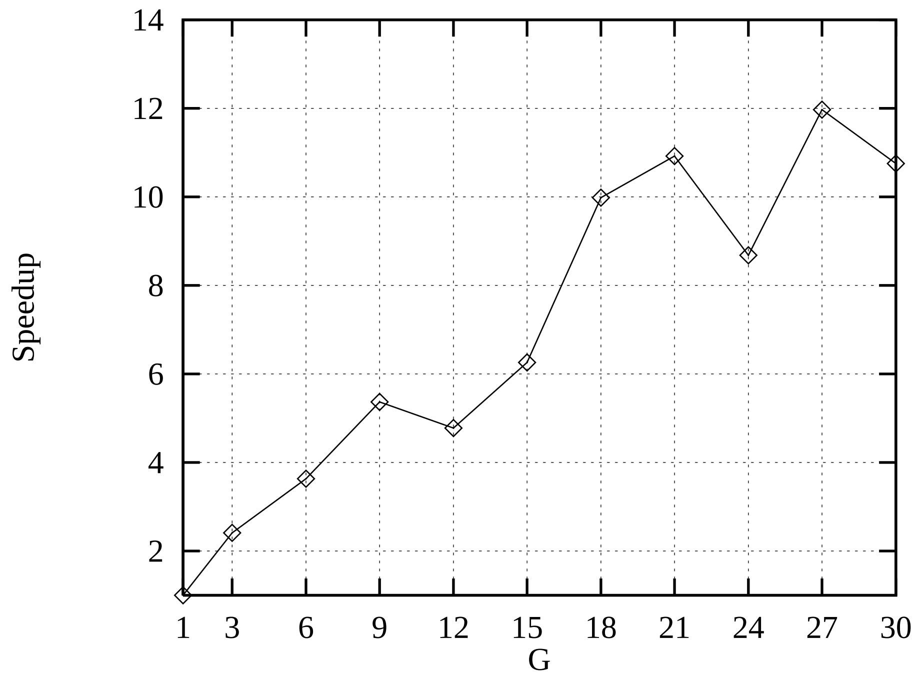


Figure 3.10: Speedup for Pentominoes benchmark for $G = 1 \ldots 30$ and $L = 21$.

3.6.1 Oracle discovery and allocation

In its first phase of execution of the 8-queens problem, PrologPF performs sequential computation for an average of 30 milliseconds to produce the 184 open oracles at the depth limit $L = 21$ (see Table 3.3). The single-cpu execution of the 8-queens problem completes in 1898 milliseconds (Table 3.1). The overhead of the initial breadth-first phases places an upper bound of 1898/30, or 63, on the possible speedup for $L = 21$.

The 184 open oracles discovered at $L = 21$ reference subtrees containing varying amounts of work. The distribution of the work expressed in milliseconds per oracle is given in Table 3.4.

Work (ms):	0	5	10	15	20	25	30	35	40	45	50	55	60
Oracles:	65	27	32	37	6	7	8	1	0	0	0	1	

Table 3.4: Oracle work distribution for 8-queens at $L = 21$

With the one-time allocation of oracles on the completion of the first phase of the breadth-first partitioning strategy, with $G = 30$ each path processor will be allocated 6 or 7 of the 184 open oracles. The combined total of the work beneath all the oracles is 2095 milliseconds, for an average oracle size of 2095/184, or 11.4, milliseconds. It should be noted that the oracle distribution contains one outlier with a subtree of 55 to 60 milliseconds, and the presence of this outlier will dominate the parallel runtime. The path processor which receives this outlier will also receive 5 other 'average' oracles, for an approximate total execution time of:

Initial breadth-first partitioning phase:	30ms
Execution of 5 'average' oracles (5×11.4ms):	57ms
Execution of 'outlier' oracle:	58ms
Approximate total runtime:	145ms

The path processor which has been allocated the outlier oracle will have the longest runtime of those in the selected group. The speedup for $L = 21$, $G = 30$ can be expected to be 1898/145, or about 13. This estimate matches the performance data listed in Table 3.3. For lower values of G, the impact of the outlier is reduced as a greater number of 'average' oracles is allocated to each path processor.

The distribution of work represented by the 864 open oracles discovered at $L = 27$ in the 10-queens problem is shown in Table 3.5.

Work (ms):	0	30	60	90	120	150	180	210	240	270	300	330	360	390	420	450
Oracles:	379	187	109	60	62	35	17	6	4	1	1	2	0	1	1	

Table 3.5: Oracle work distribution for 10-queens at $L = 27$.

With $L = 27$ the breadth-first partitioning first phase executes for 168 milliseconds producing 864 oracles. Table 3.5 shows that there are 190 oracles with underlying work greater than 90 milliseconds, permitting a reasonably even distribution across the path processors for $G \leq 30$. Unlike the 8-queens example with $L = 21$, the outliers are less significant with 864/30, or 28, oracles allocated to each path processor, with an average size of 55 milliseconds. The larger size of the problem, permitting a greater optimal L and consequent larger selection of oracles, generates more open oracles with significant subtrees, such that the speedup factor of 18 for $G = 30$ can be achieved, an improvement over the 8-queens case.

The speedup graph for the Pentominoes benchmark given in figure 3.10 shows significant reduction in parallel performance for values of $G = 12$ and $G = 24$. This reduced speedup for an increased number of available path processors is caused by an unfavourable distribution of open oracles after the breadth-first partitioning phase.

The distribution of work represented by the 848 open oracles discovered at

$L = 21$ in the Pentominoes problem is shown in Table 3.6.

Work (seconds):	0	2	4	6	8	10	12	14	16	18	20	22	24	26
Oracle count:		790	22	8	7	9	5	3	0	0	0	0	1	3

Table 3.6: Oracle work distribution for Pentominoes at $L = 21$

Table 3.12 shows there are four open oracles with subtrees resulting in searches greater than 20 cpu seconds. The oracle numbers and subtree sizes of these four largest oracles are given in Table 3.7.

Oracle Number	Subtree Size (ms)	Path Processor ($G = 24$)
183	25319	15
558	24947	6
195	24862	3
402	23982	18

Table 3.7: Allocation of largest oracles in Pentominoes problem $G = 24$.

The total workload associated with all 848 oracles discovered at $L = 21$ in the Pentominoes problem is 449583 milliseconds. This figure is greater than the single-cpu runtime because that case can be executed with $L = 1$ as no partitioning is to be performed. The average workload associated with an open oracle for the Pentominoes problem with $L = 21$ is 530 milliseconds. For $G = 24$, each path processor will receive an allocation of either 35 or 36 oracles, with an average total runtime of 18800 milliseconds. The four largest oracles each contain single subtrees larger than this average, such that the processors receiving the largest oracles might be expected to dominate the overall parallel runtime. The individual runtimes of the four longest running path processors for the Pentominoes program with $G = 24$ and $L = 21$ are given in Table 3.8.

Path Processor	Total Runtime (ms)
15	51367
3	42708
16	42701
18	36243

Table 3.8: Runtimes of four longest running path processors in Pentominoes problem $G = 24$.

The runtime of the parallel execution of the problem is determined by the longest executing path processor, in this case path processor number 15. The single-cpu runtime for the Pentominoes problem is 445959 milliseconds (see Table 3.1). Thus the speedup at $G = 24$ is 445959/51367, or 8.68 as shown in Table 3.3 and the graph in figure 3.10.

The issue of the presence of large outlier oracles is apparent in the Pentominoes problem, illustrated by the fact that path processor 15 is allocated the largest oracle (among its assignment of 35 oracles from the 848) and takes the longest time to complete. This issue is compounded with the simple one-time allocation of all the oracles at $L = 21$ to the available path processors, such that one path processor may be allocated several large oracles, while another may receive an allocation containing only oracles with small subtrees.

Figure 3.11 compares the parallel runtime performance of the Pentominoes benchmark for two similar depth limits $L = 21$ and $L = 24$. At $L = 21$ there are 848 open oracles, and at $L = 24$ there are 1410 open oracles. Although the larger count of oracles might be expected to improve the effectiveness of the work allocation, the distribution of the large oracles at $L = 24$ is particularly unfavourable to a regular round-robin allocation to the path processors.

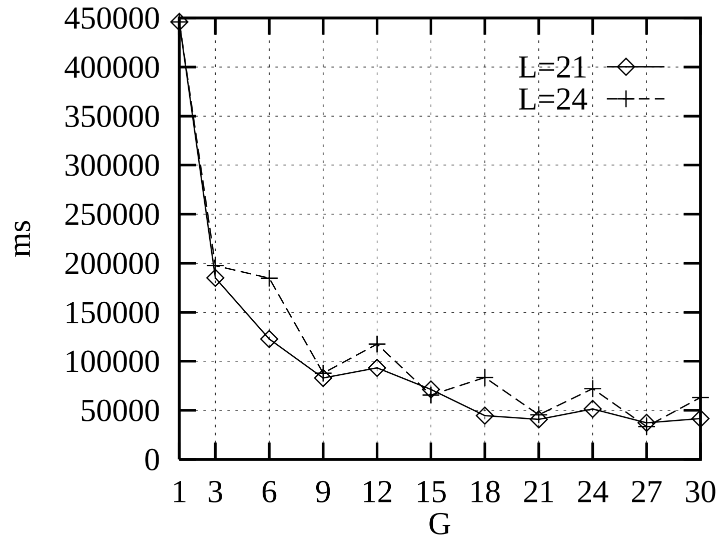


Figure 3.11: Runtimes for Pentominoes benchmark for $G = 1 \ldots 30$ and $L = 21$ versus $L = 24$.

The analysis of the oracle distributions shown in Table 3.9 shows that increasing the depth limit from $L = 21$ to $L = 24$ has not produced a significant increase in the number of large oracles.

Work (seconds):	0	2	4	6	8	10	12	14	16	18	20	22	24	26
Oracle count $L = 21$:		790	22	8	7	9	5	3	0	0	0	0	1	3
Oracle count $L = 24$:		1348	31	12	5	6	4	2	0	0	0	0	1	1

Table 3.9: Oracle work distribution for Pentominoes at $L = 21, 24$.

3.6.1.1 All-solutions versus first-solution parallel runtimes

The problem of large outliers in the list of open oracles at the selected BFP depth limit has a significant impact on the runtime of problems which require **all** the path processors to complete. This is because the runtime will be determined by the longest running path processor, and the large outlying oracles will cause the associated path processor to execute for a greater time than average.

The problem of outlier oracles with the simple breadth-first partitioning scheduling strategy is potentially less significant for problems for which only one solution is required, after which the path processors can be terminated.

The issue can be illustrated with the runtimes of the 12 processors for $G = 12$ in the Pentominoes problem (using $L = 21$). The runtimes are ordered in Table 3.10.

Path Processor	**Oracle count**	**Runtime**	**Solution Count**
0	71	2847	0
2	71	6039	0
10	70	23873	1
11	70	29037	0
8	70	29306	0
9	70	31932	0
1	71	32619	0
5	71	38048	2
7	71	42790	0
6	71	60172	4
4	71	65683	1
3	71	91939	0

Table 3.10: Path processor runtimes for Pentominoes $G = 12$, $L = 21$.

The parallel runtime to generate all the solutions is determined by path processor number 3, taking 91939 milliseconds to complete. In general, the parallel runtime is determined by whichever path processor takes the *longest* time to complete. The speedup for the all-solutions case is 445959/91939 or 4.8.

For the first-solution case, the problem will complete when the first solution is found by any of the 12 path processors. While the small number of large outlier oracles are very likely to dominate the runtime in the all-solutions case, in the first solution case a solution may be found in the average workload of a more typically loaded path processor. In fact for the Pentominoes problem with $G = 12$ and $L = 21$, the longest running path processor number 3 does not contribute *any* solutions, and the first solution is returned by path processor number 6 after 930 milliseconds.

3.6.2 Work function estimation

For the Pentominoes problem with a depth limit $L = 21$, 848 open oracles are generated. The work associated with each of these oracles can be found by performing a run of the program assigning one processor to each oracle, and logging the runtime of each path processor. The results from a simulated performance test with $G = 848$ are shown in the graph of Figure 3.12. The graph shows that there are many oracles with very small subtrees, sparsely interspersed with larger oracles.

The oracles are grouped by the amount of work in the associated subtree in the earlier Table 3.6. The 4 largest oracles in that table are given in Table 3.7 as oracle numbers 183, 195, 402 and 558. These four largest oracles can be seen in Figure 3.12.

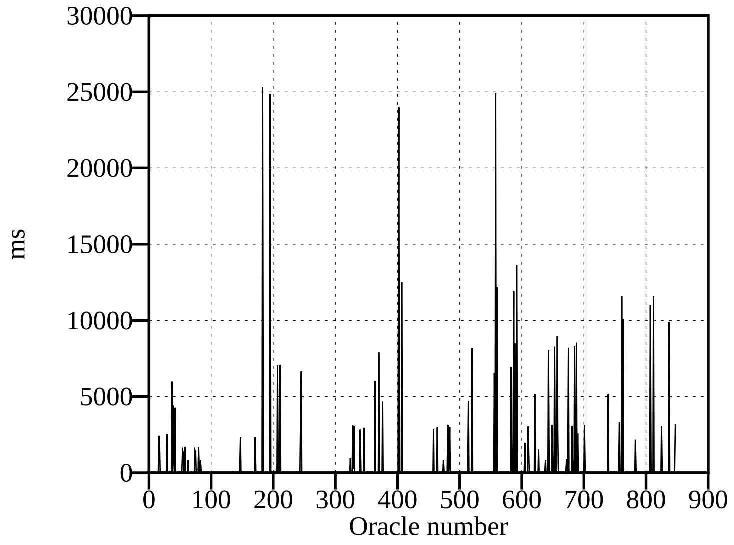


Figure 3.12: Work in milliseconds under each oracle for Pentominoes $L = 21$

A clearer picture of the distribution of work under a subset of the oracles is provided in Figure 3.13.

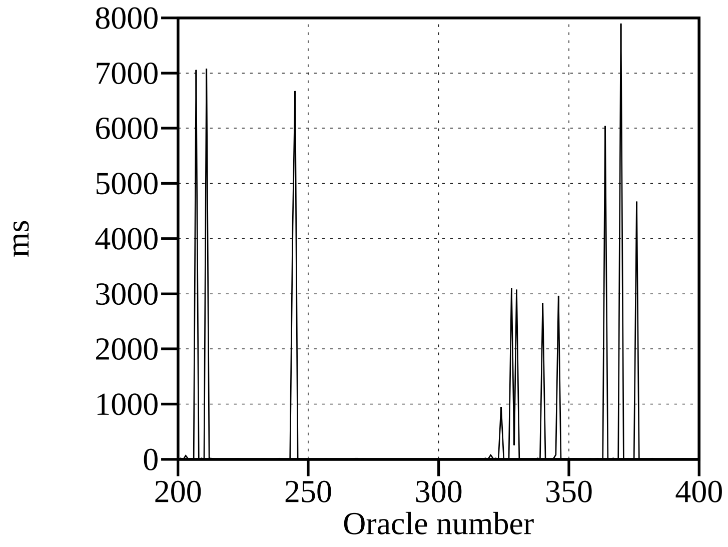


Figure 3.13: Work in milliseconds under oracle number 200 to 400 for Pentominoes $L = 21$

Earlier work by Saraswat [66] attempted to improve upon the simple breadth-first partitioning algorithm by estimating the work beneath each oracle before its allocation to a path processor. If the work estimate were accurate, then the workload could be more accurately divided between the available path processors.

The two techniques used to estimate the size of the subtrees beneath the oracles were:

1. Selective sampling of the tree.
2. Fully searching the subtrees of alternate oracles, and using the arithmetic mean of the immediate neighbours as an estimate of the work beneath the intermediate oracles.

An examination of the ordered list of open oracles discovered at $L = 21$ in the Pentominoes problem shows that the second method of work estimation would not provide useful results in that case. Table 3.6 shows that of the 848 oracles, 58 contain subtrees greater than 2000 milliseconds. The total work associated with all 848 oracles is 449583 milliseconds, and the work associated with the largest 58 oracles is 420303 milliseconds. Thus the 7% largest oracles lead to 95% of the work. These large oracles are randomly distributed throughout the ordered list of open oracles, and are most likely to have immediate neighbours which contain very little work.

3.6.3 Partitioning depth

The breadth-first partitioning strategy requires an additional parameter be passed to each path processor specifying the depth to be searched before the partitioning of the workload takes place. The selection of this parameter L can have a significant impact on the subsequent runtime of the problem:

L **too small:** not enough open oracles will be generated at this depth to provide work for all the available path processors. Thus many path processors may remain idle throughout. Any oracles generated which point to small subtrees will cause the associated path-processor to complete quickly, and then remain idle.

L **too large:** The amount of work in the tree within the depth limit may become an appreciable proportion of the total work available in the tree. The initial breadth-first partitioning phase is performed sequentially, and the parallel speedup is limited to the subsequent unlimited search phase.

The speedup performance of the breadth-first partitioning strategy with a range of values for L is given in figures 3.14, 3.15 and 3.16. The graph for the 8-queens problem includes a dashed line illustrating limits on the speedup discussed in this section.

The example of the 8-queens problem illustrates the characteristics of values of L which are too small ($L < 21$) and too large ($L > 21$). The number of open oracles discovered at each value of L is given in Table 3.11. The table also shows BF_time, which is the time taken for the initial breadth-first partitioning phase.

Table 3.11 also contains some calculated values for the minimum execution time and the maximum speedup. For values of $L = 1 \dots 12$, fewer oracles are produced than the number of available processors $G = 30$, and the maximum possible speedup is at least limited to the number of open oracles discovered. This maximum speedup assumes the oracles have subtrees of equal sizes. The actual speedup figures for $L = 1 \dots 12$ are below these maxima because the oracles are of unequal sizes. For values of $L = 15 \dots 36$, the minimum runtime of any path processor is the initial partitioning time BF_time plus the total work under all the oracles divided by G. Again this lower bound for runtime assumes the work is perfectly evenly divided among the oracles. In practice, the workload beneath the oracles is uneven resulting in lower speedups, and the actual speedup curve can be seen to fall short

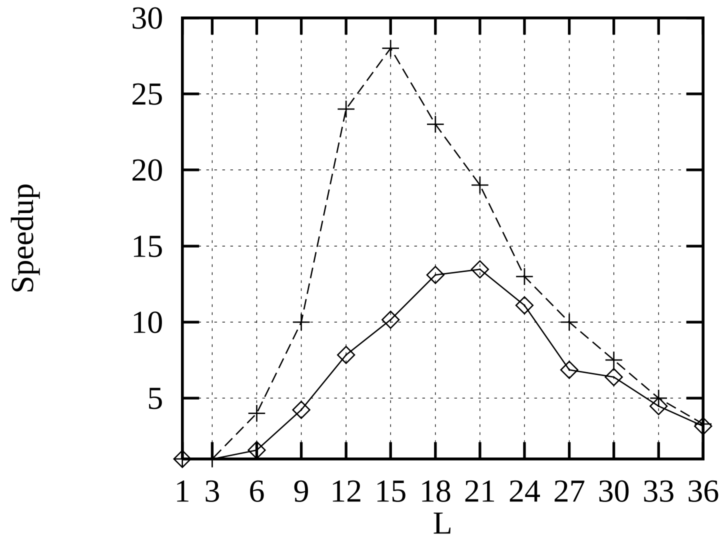


Figure 3.14: Actual and limits for speedup of 8-queens benchmark for $G = 30$ and $L = 1 \ldots 36$.

of the calculated upper bound at every point in the graph of figure 3.14. The shortfall is most significant for small values of L, where the workload is most unevenly distributed among the discovered open oracles.

The graphs shown in figures 3.14, 3.15 and 3.16 show the speedups for a range of values of L for a fixed number of path processors $G = 30$. The curves are slices through the three-dimensional graphs of L, G and speedup for each problem, given in figures 3.17, 3.18 and 3.19. The graphs illustrate the importance of the optimum value for L with one-time work allocation. The parallelisation benefits reduce either side of the optimum value for L. For the example benchmarks chosen, varying the number of processors in the group has little effect on the optimal value of L. This is because with one-time work allocation, the value of L which produces the most balanced set of open oracles is generally suitable for all groups of processors where the group size G is much less than the number of generated oracles S.

3.6.4 Fixed versus demand-based oracle allocation

Table 3.10 gives the sorted runtimes for each processor in a 12-processor configuration for the solution of the Pentominoes problem with a depth limit $L = 21$. The runtimes range from 2874 milliseconds for path processor 0, through to 91939 milliseconds for path processor 3.

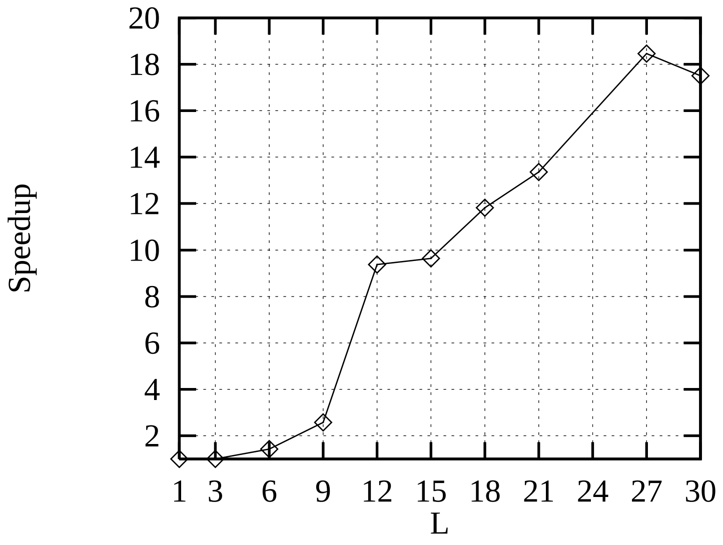


Figure 3.15: Speedup for 10-queens benchmark for $G = 30$ and $L = 1 \ldots 30$.

The total runtimes vary because the simple fixed allocation of one twelfth of the 848 oracles at $L = 21$ to each path processor (70 or 71 oracles to each) takes place without any consideration of the size of the subtree beneath each oracle. The total workload is thus randomly distributed across the 12 path processors, resulting in an uneven distribution. As was discussed in Sections 3.6.1 and 3.6.2, the distribution is particularly sensitive to the presence of large 'outlier' oracles, and in the case of the Pentominoes problem is determined by the allocation of the largest 7% of the oracles representing 95% of the work.

The fixed allocation algorithm where the 848 oracles in the Pentominoes problem are divided equally and permanently among the available path processors is particularly simplistic. If we take the case discussed above with $G = 12$ and each path processor receiving 70 oracles, a path processor may execute one or more large outlier oracles at an early stage in its execution with many oracles still to be executed. At this stage other path processors may already be idle, but the simple algorithm does not permit the reassignment of the remaining oracles to the idle path processors. This issue can be mitigated without the use of a work estimation function, as is demonstrated by the demand based algorithm discussed below.

Without an effective work estimation function for the oracles, the parallel performance might be improved with the use of a demand-based allocation algorithm rather than the current fixed distribution. The demand algorithm

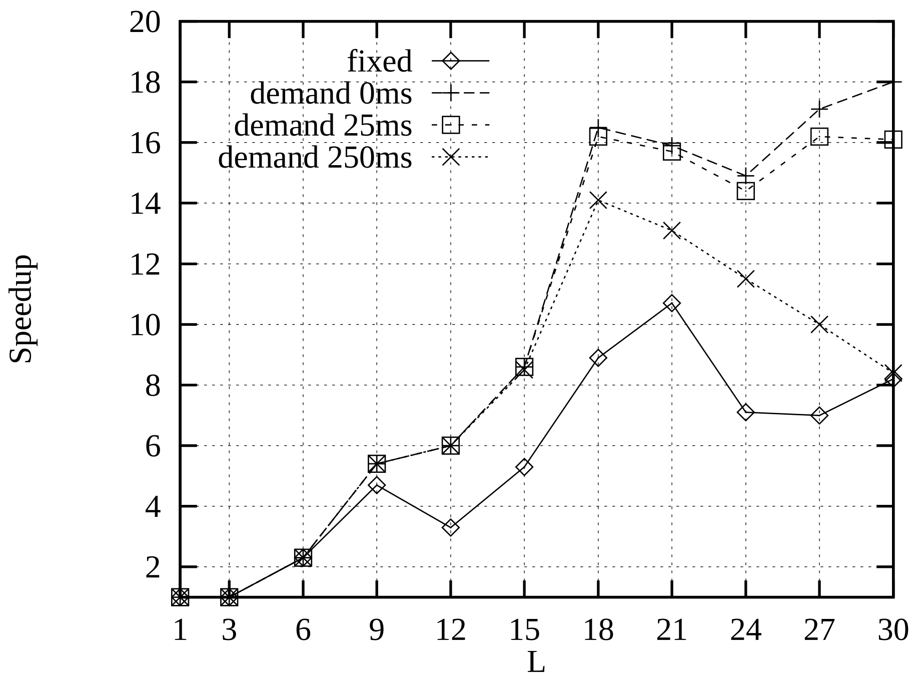


Figure 3.16: Speedup for Pentominoes benchmark for $G = 30$ and $L = 1 \ldots 30$.

would operate as follows:

1. The breadth-first partitioning phase would execute as before, generating a number S of open oracles at the depth limit L, forming an ordered list of oracles numbered $0 \ldots S-1$.
2. Each of the G available path processors with processor numbers $N = 0$ through $N = G-1$ would be assigned an initial oracle with the same number as the path processor number.
3. On completion of the search of the subtree beneath the initially assigned oracle, each path processor will request an additional oracle from the control processor. The path processor continues to request oracles after each assigned oracle is completed. Solutions found while processing an assigned oracle are returned to the control processor (or logged locally) and processing continues.
4. The control processor will allocate the remaining oracles to requesting path processors in the order of the incoming requests, until all the oracles have been allocated.

L	**Oracle count**	**Total work**	**Work per processor**	**BF_time**	**Min. Runtime**	**Max. Speedup**
1	1	1898	1898	0	1898	1
3	1	1898	1898	0	1898	1
6	4	2016	504	0	504	4
9	10	1870	187	0	187	10
12	24	1887	79	0	79	24
15	52	1948	65	4	69	28
18	102	1953	65	16	81	23
21	184	2002	67	32	99	19
24	316	2039	68	74	142	13
27	484	2168	72	113	185	10
30	720	2285	76	176	252	7.5
33	966	2428	81	300	381	5.0
36	1188	2562	85	496	581	3.3

Table 3.11: Oracle count S and oracle sizes for 8-queens with $L = 1 \dots 36$.

5. The path processors become idle on completion of an assigned oracle when the control processor indicates no further oracles are available.

For the Pentominoes problem with $G = 12$ and $L = 21$, the allocation of the 848 oracles on a demand basis can be simulated to result in the runtimes given in Table 3.12.

Path Processor	**Oracle count**	**Runtime**
10	95	33752
6	71	34007
0	29	35205
1	44	35342
4	69	35776
8	43	36268
2	79	36789
3	103	40169
7	70	41557
5	79	42527
11	105	42812
9	61	43443

Table 3.12: Runtimes of each path processor for Pentominoes $G = 12$, $L = 21$ using demand allocation with no retrieval delay.

Table 3.12 shows that the work is allocated more evenly than with the simple fixed allocation algorithm, with a widely varying number of oracles being assigned to each path processor. Path processor number 9 has the longest runtime, resulting in a speedup over the single-cpu case of 445959/43443, or 10.3, an improvement on the speedup of 4.8 with the fixed allocation.

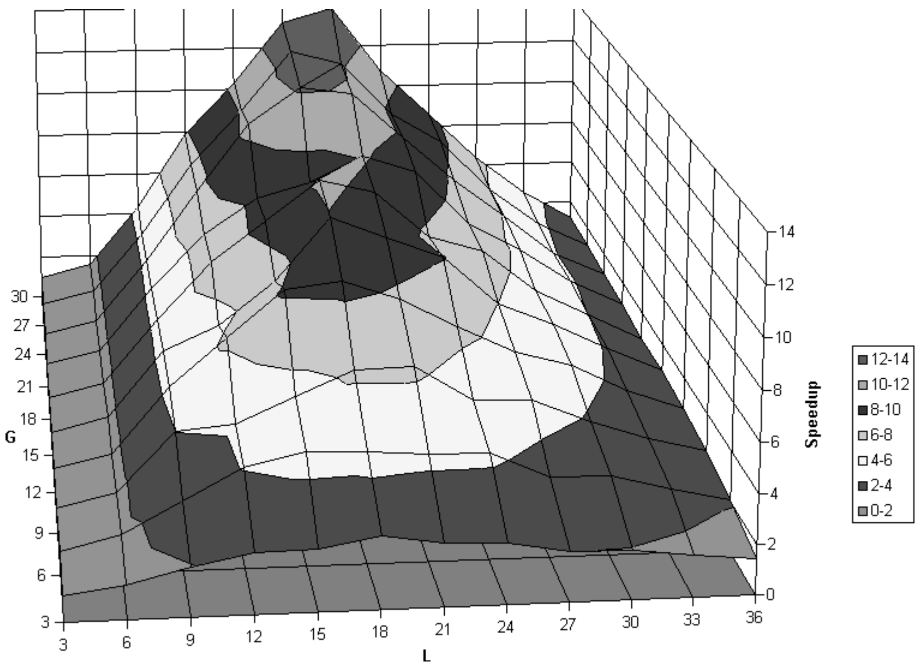


Figure 3.17: Speedup for 8-queens benchmark for $G = 1 \ldots 30$ and $L = 1 \ldots 36$.

However, Table 3.12 represents the maximum improvement over the fixed allocation algorithm as the overhead associated with each request for an oracle has been set at zero milliseconds. The demand allocation algorithm is sensitive to the communications delay associated with every request for an oracle, as can be shown with the Pentominoes problem for a range of depth limits with $G = 30$.

Figure 3.16 shows the speedup achieved by PrologPF for the Pentominoes problem for values of the depth limit $L = 1 \ldots 30$ and $G = 30$ using the fixed oracle allocation algorithm. The graph also shows the calculated speedup performance for a demand-based oracle allocation algorithm with oracle retrieval delays of 0, 25 and 250 milliseconds.

At each value of the depth limit L, the number of oracles generated S and the time taken for this initial oracle discovery phase are given in table 3.13.

With reference to Figure 3.16, the demand-based oracle allocation algorithm with the retrieval delay of zero or 25 milliseconds outperforms the fixed allocation algorithm used by PrologPF for all values of L. With an oracle retrieval delay of 250 milliseconds, the overhead of the demand-based algo-

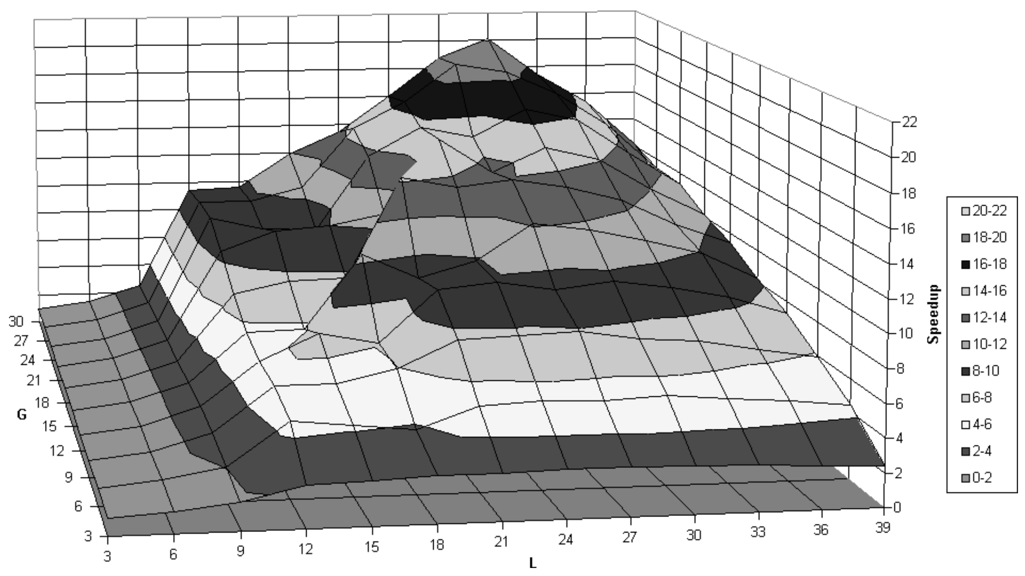


Figure 3.18: Speedup for 10-queens benchmark for $G = 1 \ldots 30$ and $L = 1 \ldots 39$.

rithm increases with the number of oracles S. Table 3.13 shows that for $L = 30$, the number of oracles discovered $S = 3396$. For $G = 30$ at least one processor will be allocated at least 3396/30, or 113 oracles. The time taken to retrieve these 113 oracles places an upper limit on the possible speedup with the demand-based allocation algorithm. With a retrieval delay of 250 milliseconds, the cumulative delay associated with the allocation of the oracles will be 113×250, or 28250 milliseconds, such that at least one path processor will take at least this amount of time to complete, so the speedup is limited by the retrieval delay to a maximum of the single-cpu time divided by 28250, or 15.8. In practice the oracles also have a subtree search time, such that the speedup curve peaks for a value of L with an optimal balance of granularity of work under the discovered oracles with a sufficiently small oracle count S to mitigate the retrieval overhead. The graph in Figure 3.16 shows this balance to be optimal with $L = 18$ for an oracle count $S = 472$.

3.6.5 Work splitting

After the initial breadth-first partitioning phase in which the open oracles are discovered at the selected depth limit L, PrologPF uses a simple fixed allocation algorithm to assign the oracles to the available path processors. After this assignment takes place, with one-time partitioning the path pro-

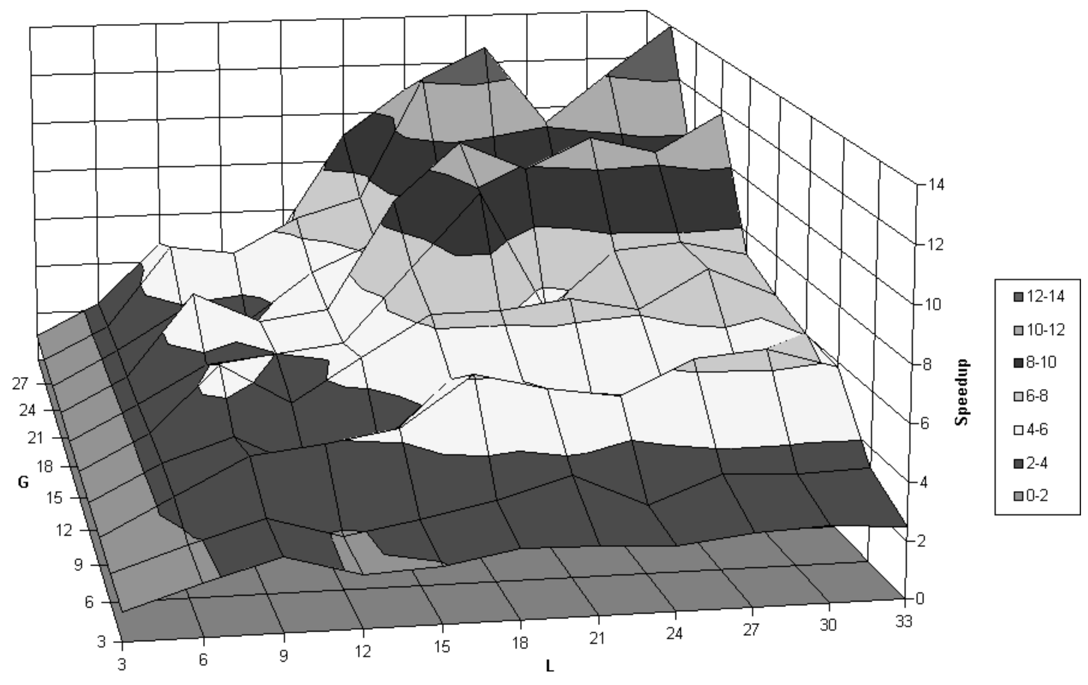


Figure 3.19: Speedup for Pentominoes benchmark for $G = 1 \ldots 30$ and $L = 1 \ldots 33$.

cessors search the subtrees of all their allocated oracles and become idle when all this work is completed. Some path processors will complete their assigned work and become idle before others. An example of this behaviour is illustrated in Table 3.10, where path processor 0 becomes idle after only 2827 milliseconds while path processor 3 executes for 91939 milliseconds.

As the parallel performance is determined by the path processor with the highest runtime, a more balanced distribution of the work will increase the overall speedup. To balance the workload in the example discussed, work must be transferred from path processor 3 to other path processors.

Relative to the fixed allocation of oracles used by PrologPF and DelphiKS, the demand allocation of oracles discussed in Section 3.6.4 represents a dynamic distribution of the workload. In effect the work that would otherwise reside in the pool of oracles assigned to processor 3 is collected by other path processors as they become idle. However, the 'outlier' oracles discussed in Section 3.6.1 represent an additional problem, in that a path processor will proceed without interruption to search the entire subtree beneath any given oracle.

Depth Limit L	**Oracle count S**	**BFtime**
1	1	0
3	2	4
6	12	4
9	44	12
12	134	58
15	268	156
18	472	332
21	848	672
24	1410	1261
27	2256	2281
30	3396	3702

Table 3.13: Oracle count S and initial oracle discovery time (BFtime) for Pentominoes problem with $L = 1 \ldots 30$.

This issue is more significant the more imbalanced the work between the S discovered open oracles at the selected depth limit L. Empirically, the imbalance in workload between the discovered oracles become worse the smaller the depth limit and the fewer the discovered oracles.

The issue of work splitting is illustrated in Figure 3.20.

In Figure 3.20 the initial depth limit has been set at L, and the illustrated path processor N has been assigned the open oracles A, B, C, D and E. The figure illustrates the situation at some point while path processor N is searching the subtree of oracle C, having completed oracles A and B. At this point other path processors have completed all their oracles and have become idle. The parallel performance of the system may be improved if the remaining work of path processor N can be assigned to the idle path processors. With the architecture of the Delphi machine, the remaining work can be categorised as being of two types:

1. The oracles D and E represent subtrees yet to be searched.
2. Path processor N is currently at point X in its search of the subtree beneath oracle C, proceeding with the standard Prolog depth-first left-to-right search. The portion of the subtree under oracle C to the right of X remains to be searched.

For the first case, the reassignment of the work of the remaining oracles can be effected by allocating the remaining oracles D and E to two idle processors.

The second case, in which path processor N has partially searched the subtree under oracle C to arrive at point X, may require more complex treatment involving an incremental breadth-first partitioning phase starting from

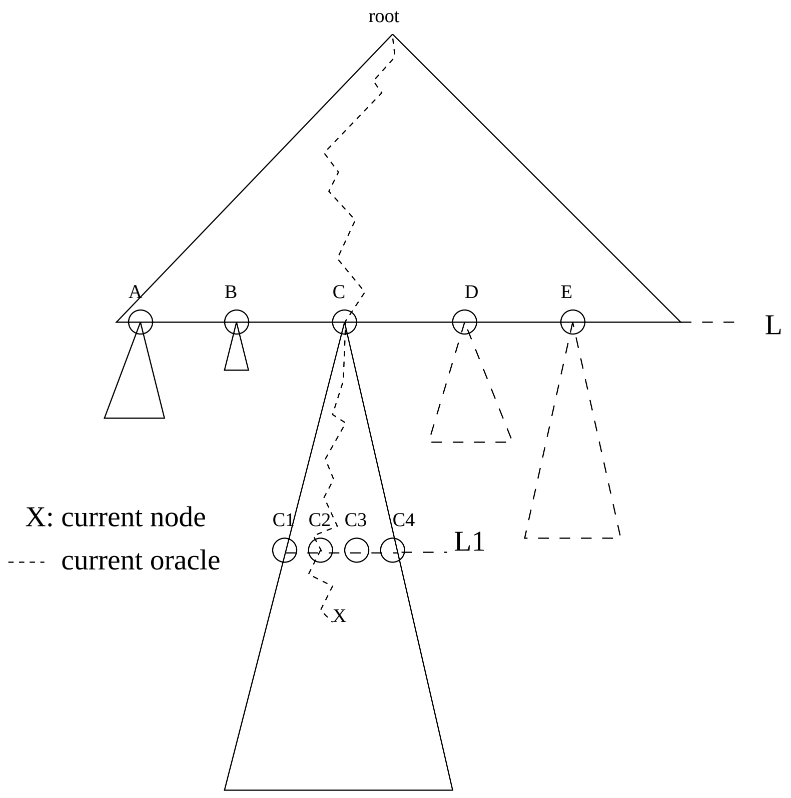


Figure 3.20: Work splitting at busy path processor.

oracle C rather than the root of the problem search tree. This phase will discover the open oracles C1, C2, C3 and C4 at the new depth limit L1.

Then the simplest approach may be to discard the work already performed by path processor N, and to add that path processor to the pool of idle path processors for the oracle reassignment. Those processors then search the subtrees below oracles C1, C2, C3 and C4 in parallel.

An issue arises if solutions have been found by path processor N and returned to the control processor. Further solutions found by the group of processors assigned oracles C1, C2, C3 and C4 may be duplicates of those already returned. If the previous search to position X by path processor N is ignored, then the solutions returned must be labelled with their associated oracle, such that subsequent duplicates can be discarded. The use of an oracle to uniquely identify a point in the search tree provides a powerful mechanism to label any solution with a unique identifier. In problems requiring only one solution, or where duplicate solutions are acceptable or ignored, the transmission of the oracle with the solution is no longer necessary.

A more complex approach to reassigning the work of path processor N beneath oracle C in Figure 3.20 can take advantage of the availability of the current oracle in path processor N referring to point X at the time of interruption. If the incremental partitioning depth L1 is smaller than the current depth of X, then the oracles found at L1 have the following characteristics:

1. One of the open oracles discovered at the incremental depth limit L1 (C1, C2, C3 or C4) will form a prefix of the oracle to X. This is necessary because if X is a valid point in the search space then all oracles representing every choice point up to X *must* have further open branches to include X, and cannot be closed oracles representing success or failure. For the following discussion this oracle will be called C_x.
2. All oracles at the incremental depth limit L1 which are numerically smaller than the oracle formed from the first L1 indexes in the oracle leading to X have already been fully searched by path processor N. This is true because the oracle indexes are ordered in the same order as the textual sequence of the associated clauses, and the search order used by PrologPF is the same as that of Prolog, namely top-down.

If further refinement is required, the subtree referred to by oracle C_x (see above) can be further partitioned at an incremental depth L2 to generate additional open oracles. The same reasoning can be used as above to identify the single oracle within this ordered list which includes X, such that all open oracles to the right of that oracle can be assigned to idle path processors.

Chapter 8 builds upon a modified breadth-first partitioning technique to support work-splitting in PrologPF.

3.7 Breadth-first partitioning versus stream-AND parallelism

Stream-AND parallelism in logic programming relies upon the treatment of certain relations as *producers* and other relations as *consumers*. Newmarch provides a tutorial introduction to stream-AND parallelism in [59] Chapter 10, and the technique is reviewed in this dissertation in Chapter 2 Section 2.1.2.

A conjunctive goal such as `member(X,Biglist), factorial(X,Y)` can be viewed as a pair of connected parallel processes `member` and `factorial`. The `member` process generates values and sends them to the `factorial` process, which in turn produces the associated sequence of factorials. The parallel speedup arises from the fact that the `member` process can be working on the production of the next value while the `factorial` process is still working on the previous value.

The breadth-first partitioning scheduling strategy can produce comparable benefits when the producer process involves relatively little work. This can be illustrated with a sample tree for a `member/factorial` query with the program listed below. The search tree for the following `member/fact` program is given in Figure 3.21.

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).

fact(1,1).
fact(X,F) :- X > 1, X1 is X-1, fact(X1,F1), F is X*F1.

:- member(X,[4,3,2,1]), fact(X,Y).
```

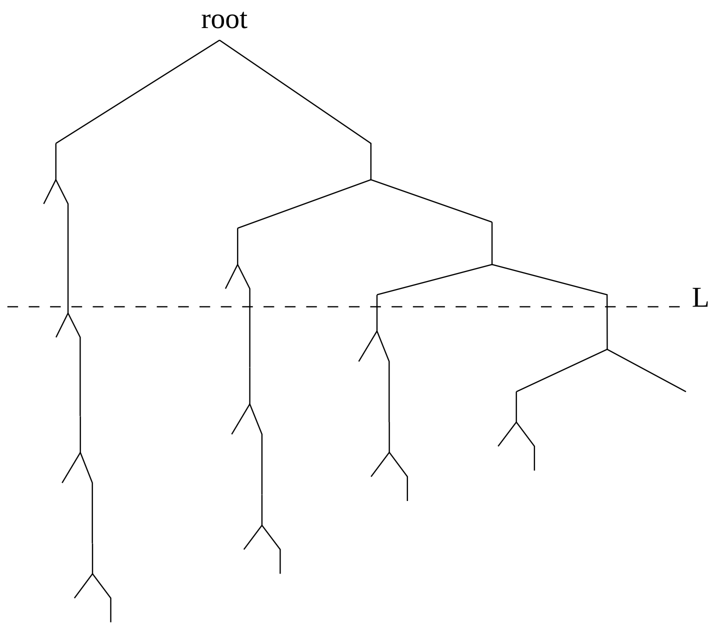


Figure 3.21: Search tree for `member/fact` program.

At the selected depth limit L a number of oracles are discovered which can be passed to available path processors for execution. These path processors will recompute the value of `X` generated by the `member` relation by following the assigned oracle. Following an oracle is an efficient operation which takes a time proportional to the length of the oracle, L. The reconstructed value of `X` is then passed to the `fact` relation for the relatively lengthy computation. The approach is most effective if the subsequent computation is much greater than the initial work generating and following the open oracles, and the technique is suited to *generate-and-test* programs where the *test* is complex.

3.8 Conclusions

The benchmark programs used to evaluate the pure Prolog performance of PrologPF have sufficient OR-parallelism to provide useful results in tests with 1 to 30 path processors[2]. Execution of the programs has illustrated issues relating to the overall size of the selected problem, the use of the breadth-first partitioning scheduling algorithm, and the selection of the depth limit used in that scheduling algorithm.

PrologPF has an efficient implementation with the use of simple oracle management primitives `o_build` and `o_follow` written in 'C' combined with a standard Prolog compiler. Similar primitives can be implemented in Prolog giving a flexible system suitable for evaluation of the use of oracles with an acceptable performance penalty. The PrologPF overhead relative to the use of the standard compiler in a single-cpu environment for the test programs was 9 to 19%.

The breadth-first partitioning algorithm provides a means of allocating the work to a number of available path processors with no subsequent communication after the initial assignment of the problem. The optimal selection of the depth limit for the breadth-first partitioning strategy has a significant impact on the effectiveness of the subsequent parallelisation, with the fixed one-time work allocation algorithm resulting in an uneven utilisation of the path processors. The utilisation of the path processors would be more even if the breadth-first oracle discovery technique were complimented with a demand-based allocation algorithm and work splitting of large oracles during their execution if other path processors are idle.

PrologPF with its breadth-first partitioning algorithm provides similar parallelisation benefits to those obtained with stream-AND parallelism for generate-and-test programs in which the generation procedure is relatively lightweight.

[2]The experimental system at Cambridge was generally limited during this research to 30 processors

3.9 Summary

Three OR-parallel pure Prolog benchmarks have been used to investigate the effectiveness of the parallelisation techniques used in PrologPF:

8-queens: the chess-based problem of fitting eight queens onto an eight-by-eight chessboard with none attacking any other.

10-queens: the same problem with ten queens and a ten-by-ten chessboard.

Pentominoes: the problem of fitting twelve defined geometric shapes onto a three-by-twenty board.

The programs selected have been used to:

1. Assess the speedup provided by PrologPF for the execution of each problem on a range of distributed systems with 1 to 30 processors.
2. Compare the performance of PrologPF with previous implementations of the Delphi machine.
3. Compare the performance of PrologPF and previous Delphi implementations with the single-cpu performance of other compilers.
4. Investigate the overhead of the partitioning time for a range of values of the depth limit parameter of the breadth-first partitioning algorithm.
5. Investigate the impact of the random distribution of the workload beneath the typical range of discovered open oracles.

The earlier implementation of a Prolog system using oracles, DelphiKS, utilised an extended Warren Abstract Machine [73] with additional instructions for oracle manipulation. A prototype version of PrologPF uses Prolog data structures to hold the oracles, with Prolog procedures to create and interpret the structures. The final version of PrologPF uses similar procedures written in 'C' with the oracles held in 'C' arrays for a faster execution. Each system has produced compiled binaries suitable for parallel execution on a distributed system of common Unix workstations.

PrologPF is based upon the 'wamcc' Prolog compiler [30]. The following table compares the single-cpu performance of DelphiKS, PrologPF with Prolog oracle primitives, PrologPF with C primitives, wamcc, and SICStus Prolog Version 3 (compactcode) on the same DECStation 3100. The figures are calculated from the geometric mean of the results of the three benchmarks used, and are normalised against wamcc.

DelphiKS	**PrologPF(Prolog)**	**PrologPF(C)**	**wamcc**	**SICStus**
19.8	3.95	1.15	1.00	1.61

Table 3.14: Single cpu runtime ratios.

In this analysis PrologPF uses a breadth-first partitioning strategy, in which an initial phase of execution proceeds in the normal Prolog depth-first left-to-right manner, but is limited to a selected AND-OR depth. During execution, PrologPF maintains a list of the indexes of each clause leading to the current position in the search, this list being the current *oracle*. Each time the depth limit is reached in the initial phase, the current oracle is recorded on the *oracle stack*. On completion of the initial phase of execution, the oracles on the oracle stack are assigned to the available path processors for re-execution in a second phase in which the subtree beneath each can be searched.

For the 8-queens problem, PrologPF achieved a maximum speedup with 30 processors of 13.5. On the same distributed system the speedup achieved for the 10-queens problem was 18.5 and for the Pentominoes problem was 10.8. The runtime for the 8-queens problem was reduced from 1.9 seconds to 141 milliseconds, the 10-queens problem from 46.5 seconds to 2.5 seconds, and the Pentominoes problem from 446 seconds to 41.5 seconds.

The benchmark programs were used to investigate the characteristics of the breadth-first partitioning strategy. In general, the actual speedups achieved were less than the number of available path processors. The analysis of the actual performance of the benchmarks clarified the issues involved, suggesting improvements to the technique.

The issues can be summarised as follows:

1. The initial breadth-first partitioning phase of execution is performed sequentially, without benefit of parallel speedup. The time taken to generate the open oracles during this phase places an upper bound on the overall speedup. This issue is most apparent in programs with relatively small search trees, as the initial breadth-first phase can represent an appreciable proportion of the overall execution.

2. After the open oracles have been generated and allocated, each path processor will follow each of its assigned oracles to arrive at the associated subtree. Following an assigned oracle represents a recomputation of that section of the search space. If a large number of oracles is generated, each of a significant length (the depth limit was relatively deep) then this recomputation overhead may be significant.

3. With a fixed one-time allocation of oracles to the available path processors without any consideration of the work beneath each oracle, the work to the path processors may be imbalanced, leading to a range of total runtimes for the path processors. As the parallel performance for an all-solutions problem is determined by the longest running path processor, this leads to a reduction in overall speedup.

4. The analysis of the benchmarks shows that the subtrees represented by the oracles discovered at the depth limit refer to subtrees containing widely differing amounts of work. In many cases, the majority of the oracles refer to relatively small subtrees, while a minority refer to a small number of very large subtrees. Not only all the oracles allocated at the end of the first breadth-first partitioning phase, but the subtree beneath each individual oracle is searched to completion. Other path processors may be idle while one path processor is busy in the search of a very large subtree.

The analysis of these issues suggests a number of improvements needing further work:

- The time spent in the initial breadth-first partitioning phase is a more significant issue for small problems, becoming less significant as the problem size increases. This issue might be ignored for large problems, or compile-time analysis might be used to optimise the strategy for smaller problems such as 8-queens.

- The recomputation overhead associated with following oracles is very low with the breadth-first partitioning strategy, as the time spent following the allocated oracles to arrive at the designated subtree is much lower than the time spent searching the subtrees. For this overhead to be minimised a good technique is needed to generate as many oracles with non-trivial subtrees as possible. The automatic partitioning strategy [49, 66] was greatly impacted by this issue, and breath-first partitioning improves the situation by generating a much larger pool of open oracles.

- The fixed allocation of the oracles after the initial breadth-first partitioning phase minimises the subsequent communication requirements, but leads to a significant imbalance of the workload. The workload can be much better balanced with a demand-based strategy, in which the path processors request additional oracles on completion of the search of each assigned oracle. A trade-off is made of the workload balancing against the communications requirements.

- The presence of large outlier oracles in the pool discovered at the selected depth limit implies that work splitting is required to interrupt

> the execution of a busy processor to re-partition the work associated with the large oracle. One approach is to re-use the breadth-first partitioning strategy within the subtree represented by the large oracle. The ordering of the clause indexes combined with the Prolog strict top-down left-to-right search means that some optimisations can be applied to take advantage of the search within the subtree already performed by the busy path processor. These enhancements are discussed in Chapter 8.

For some problems, PrologPF with the breadth-first partitioning strategy provides similar parallel behaviour to that obtained with stream-AND parallelism. The producer-consumer model in stream-AND parallelism is replaced in PrologPF with a recomputation technique for the generation of the 'produced' values. The techniques are comparable in efficiency if the computational requirements of the producer procedure are relatively low.

In general, PrologPF provides effective speedup for many Prolog programs on a distributed network of general-purpose workstations. The technique is simple, and suited to an environment where many workstations are available. No special programming techniques are required, and the use of oracles allows speedy recovery in the event of an individual workstation failure.

Chapter 4

Cuts versus the Delphi Principle

This chapter discusses the role of the Prolog extra-logical operator *cut* in pruning the sequential search tree, and highlights the issues that arise when the tree is searched in parallel. Alternative approaches to addressing the issue are considered. Gupta and Santa Costa discuss the related issues with AND-OR parallel Prolog programming in [40].

4.1 Cut in standard Prolog programs

Standard Prolog [35] provides a built-in extra-logical predicate "cut" (`!/0`) aimed at performing some control by pruning the search tree. The predicate always succeeds, but has a drastic effect on the sequential search tree: some branches are pruned, removing the associated sub-trees, in order to force a predication to execute more quickly without constructing and visiting those sub-trees. The example program containing *cut* from [35] is given below:

```
p(X,Y) :- q(X), !, r(X,Y).
p(X,Y) :- s(X).

q(a).
q(b).
q(c).

r(b,b1).
r(c,c1).

s(d).

:- p(U,V).
```

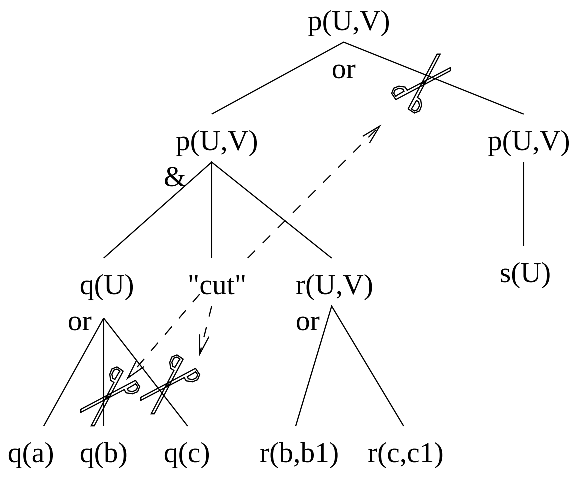


Figure 4.1: The effect of *cut* on the AND-OR search tree.

The AND-OR search tree for this simple example is given in Figure 4.1.

The search tree can be transformed to the OR-only tree given in Figure 4.2, more clearly representing the execution sequence of the Prolog depth-first left-to-right search. The first "cut" encountered prunes the alternative branches to the right of the current branch, such that only one clause is used to solve `p(U,V)` and also only one clause for `q(U)`.

The use of *cut* by a Prolog programmer requires careful analysis of the sequence in which the search tree is traversed, in order to ensure the required behaviour is obtained. The built-in predicate is often used to enforce determinacy of a relation, usually within a particular mode of call. A typical example can be found in the definition of the relation `max`:

```
max(X,Y,X) :- X > Y, !.
max(X,Y,Y).
```

The `max` relation is expected to be used with the first two arguments instantiated, and the third a variable. On succeeding, the third argument will be unified with the larger of the first two arguments. The use of cut assumes the top-down, left-to-right execution of sequential Prolog: *if* the first clause succeeds, *then* the alternative clause for `max` will be pruned from the search. If the cut were omitted, a goal such as `:- max(5,3,X)` would return two answers, {`X=5`, `X=3`}, and to avoid this problem the relation would be defined with additional conditional guards:

```
max(X,Y,X) :- X > Y.
max(X,Y,Y) :- X =< Y.
```

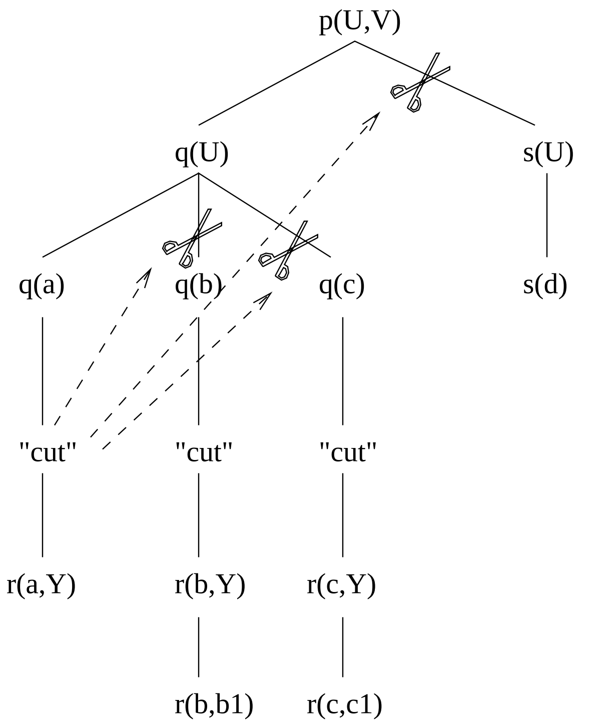


Figure 4.2: Effect of *cut* on OR-only search tree.

Thus the presence of cut has the following implications:

1. Subsequent conditions can be omitted from later clauses in the procedure.

2. The Prolog system can execute more efficiently by avoiding the construction of choice points, unifying the arguments and calling the guard conditions of later clauses.

3. The first definition of `max` which would otherwise return multiple solutions is rendered deterministic.

While the relation `max` has the required behaviour in the expected mode, the non-logical definition of the relation can introduce problems. If the relation is called with all three arguments instantiated then unexpected results may be produced. For example, with the first definition of `max` with cut, the goal `:- max(5,3,3)` will *succeed*, which might not be what the programmer expected. On unification of the arguments with the head of the first clause, the subgoal `X > Y` *fails*, such that the cut is never reached, and execution continues with the second clause which *succeeds*.

While the use of cut in procedures such as `max` is a common programming practice in Prolog, the problems caused can be avoided with the logical definition given in the second example. The deterministic execution of a procedure can be made explicit through the use of functions, as discussed in detail in Chapter 5, without the use of non-logical relations with cut.

4.2 Experimental Analysis

The benchmarks used in the earlier analysis of the PrologPF system contain a number of deterministic relations. The efficiency of execution can be improved by the addition of cuts to the programs:

8 queens and 10 queens: The program used for these benchmarks contains two deterministic procedures, `notthreatened` which succeeds if two pieces are not attacking each other and `safe` which succeeds if a partial board passed as an argument has no attacking pieces. One cut can be inserted into each procedure to improve the efficiency of the deterministic execution.

Pentominoes: This benchmark contains six deterministic procedures, used to initialise the board and test the partially filled board for consistency. The introduction of six cuts, one per procedure, makes the determinism explicit to the Prolog system for a more efficient execution.

For the execution of the benchmark programs on a DECStation 3100, Table 4.1 shows the frequency of cuts encountered during the search of each proof tree.

Benchmark	Source Cuts	Execution Time(ms)	Cuts Encountered	Time per cut
8-queens	2	1898	28666	66μs
10-queens	2	46497	814772	57μs
Pentominoes	6	445959	197878	2.5ms

Table 4.1: Count of cuts encountered for the benchmarks on a single cpu.

Table 4.1 gives the total number of cuts encountered for each benchmark during the traversal of each search tree. Using the execution times for the single-cpu case (Table 3.1 in Chapter 3), the average period between each cut can be calculated. In a parallel processing case this period represents an upper bound on the average time between the discovery of each cut in the search tree, and the average will reduce by the speedup factor of a parallel execution.

4.3 Strategies for dealing with *cut* in the Delphi Machine

The breadth-first partitioning algorithm can be applied to the search tree represented in Figure 4.2. If the partitioning depth limit is set at a low figure, such as $L = 2$, then oracles will be created for the alternate paths via `q(U)` and `s(U)` in the diagram. If these oracles are allocated to different path processors, then the discovery of the cut after the solution of `q(a)` must be communicated to the path processor executing the other oracle, such that that search can be aborted.

The path processor searching the subtree to be pruned on the discovery of the cut may have already communicated solutions to the control processor.

If the system is intended to execute Prolog programs in parallel including subtrees which may be pruned by cuts discovered by other path processors, then the simple Delphi system must be extended as follows:

- Solutions reported to the control processor must be tagged with the associated oracle.
- Any subsequent reporting of a solution to a client program must be delayed until the subtrees to the left of the solution in the OR-only tree have been fully searched, to ensure the absence of cuts which would otherwise have pruned the solution.
- Similarly, the reporting of solutions found within the depth limit during the first phase of the scheduling algorithm must be delayed until the left subtrees have been searched.
- The discovery of a cut in a subtree must be communicated to the path processors searching subtrees to the right of the discovered cut, such that pruning can be applied. However, the communication must be delayed until the subtrees to the *left* of the discovered cut have been fully searched to ensure that the subtree containing the cut should not itself be pruned.

The pruning of the search tree is limited to the depth of the clause containing the cut. The example in Figure 4.2 can be extended with an additional procedure:

```
t(X,Y) :- p(X,Y).
t(a,b).
```

Figure 4.3 shows the search tree for the query `:- t(U,V)`. The pruning due to the cut in the OR-only tree beneath `q(a)` is limited to subtrees beneath the node labelled `p(U,V)`.

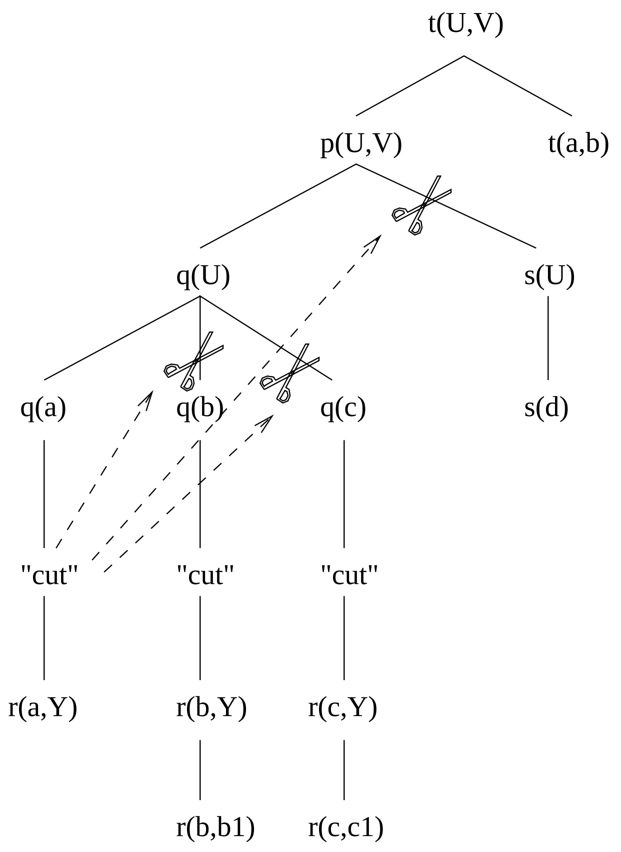


Figure 4.3: Affect of *cut* limited to depth of containing clause in OR-only tree.

In communicating the discovery of the cut to other path processors, an *oracle* can be used to define the choices affected by the cut. With reference to Figure 4.3, on discovering the cut beneath `q(a)` the path processor must communicate the oracle referring to the node labelled `q(U)`. Path processors receiving the communication must abort any search of subtrees with a prefix equal or greater to that oracle. Additional data must be recorded to relate each cut to the root of the correct subtree in the OR-only tree such that the required oracle can be created.

Section 4.2 shows cuts may be encountered during the execution of a program thousands or even millions of times each second. The target architecture for PrologPF assumes a large supply of processing power with a relatively limited communications capacity. For this reason distributed support for cut was not implemented, and an alternative approach taken.

4.4 Cuts in PrologPF

The requirement for "cut" in PrologPF is accommodated in two ways:

1. Support for higher-order functional programming is provided to mitigate the need for cut to provide:
   - Efficient deterministic execution of relations which would otherwise return multiple solutions. These relations should be replaced with functions in PrologPF. Functional support in PrologPF is discussed in detail in Chapter 5.
   - Support for the use of boolean functions, where otherwise the programmer might have used the more error-prone negation-as-failure.
2. Cut is permitted in user procedures, but those procedures *must* be deterministic. Oracle support is suspended during the execution of procedures containing cut.

The requirement that the relations containing cut must be deterministic is illustrated by the search tree for the following program given in Figure 4.4.

```
t(X,Y) :- p(X,Y), q(X).
t(a,b).

p(X,Y) :- q(X), !, r(X,Y).
p(X,Y) :- s(X).

q(b).
q(c).

r(b,b1).
r(c,c1).

s(d).

:- t(U,V).
```

The example program has one solution for the subgoal `p(U,V)`: {`U=b, V=b1`}. Oracle processing is suspended during the execution of that subgoal due

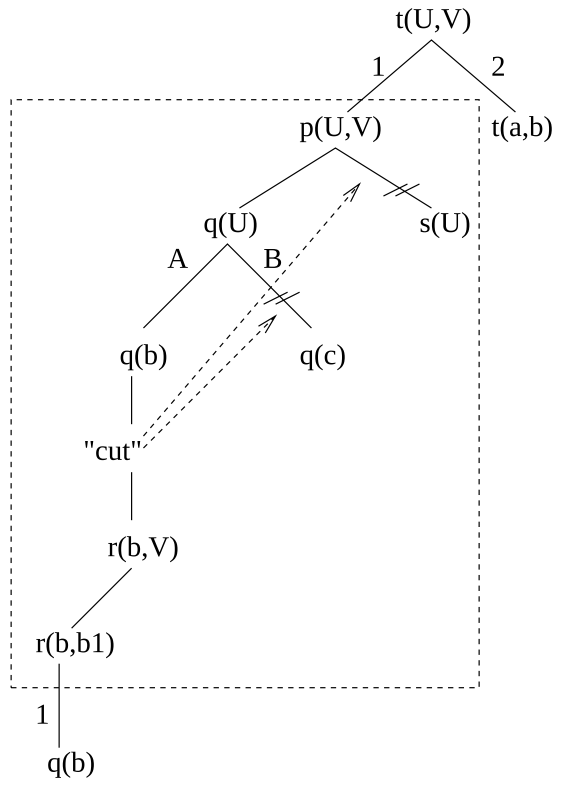


Figure 4.4: Deterministic execution of a PrologPF relation containing *cut.*

to the compiler detection of the cut in the procedure. When the relation succeeds with its single solution, oracle processing continues such that the solution to the top-level query has the oracle *[1,1]*.

Figure 4.5 gives the search tree for the same program with the addition of the clause `r(b,b2)` at the end of the procedure `r`, which becomes:

```
r(b,b1).
r(c,c1).
r(b,b2).
```

The additional clause for `r` means that the subgoal `p(U,V)` has two solutions {`U=b, V=b1`} and {`U=b,V=b2`} in spite of the presence of the cut within the procedure for `p`. Oracle processing is suspended during the execution of the subgoal `p(U,V)`, and restarted on the success of that subgoal. The multiple solutions to `p(U,V)` result in two positions in the search tree having the same oracle. The oracle *[1,1]* now refers to the position of both solutions

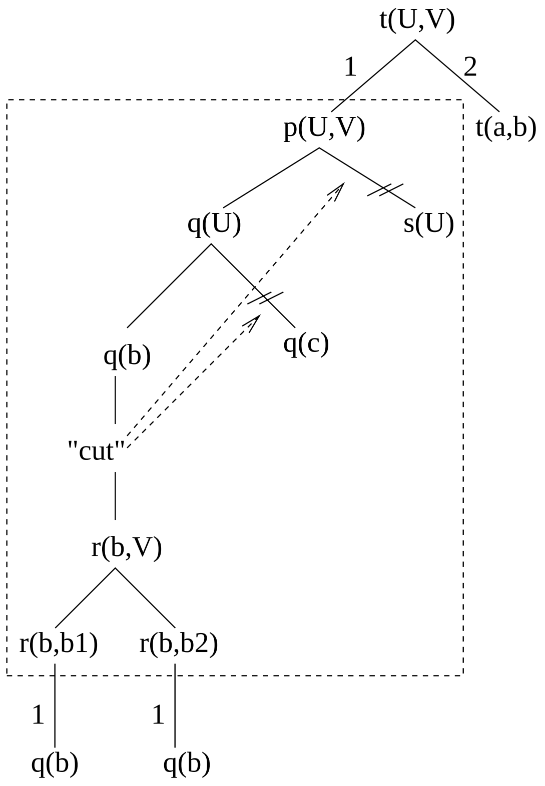


Figure 4.5: Oracle ambiguity cause by *cut* in a relation with multiple solutions.

in the search tree, the ambiguity preventing the use of oracles for parallel partitioning.

It is important to note that the incompatibility of oracles with the use of cut arises from the subsequent use of open oracles in the partitioning of the workload among the distributed path processors, illustrated with the same example in Figure 4.6.

In Figure 4.4, at the depth limit L selected, there are three open oracles: *[1,1,1]*, *[1,1,2]*, and *[1,2,1]*. The second and third open oracles refer to subtrees that should be pruned on discovery of the cut in the subtree referenced by the first oracle. Suspending oracle processing during the execution of deterministic procedures containing cut ensures that no open oracle can be generated within that procedure, potentially referring to a subtree that would otherwise be pruned by a cut elsewhere in the code. With the breadth-first partitioning strategy used in PrologPF, each open oracle is discovered at the fixed depth limit. The issue of oracle ambiguity caused by non-deterministic

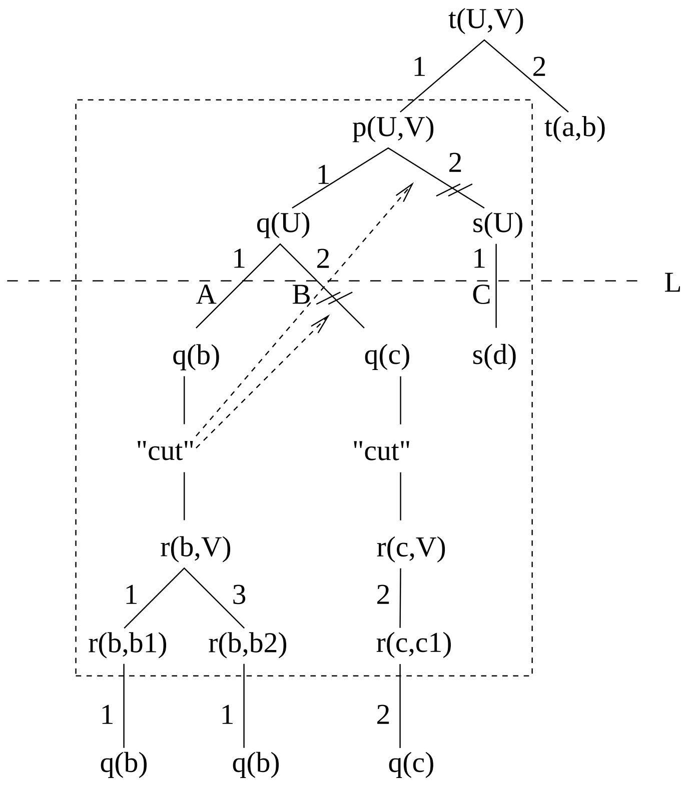


Figure 4.6: Pruned subtree allocation caused by open oracles in relation with *cut*.

relations containing cut would be addressed by an extended definition of the depth limit:

- Oracle processing continues during the execution of procedures containing cut.
- Forced failure at the selected depth limit in the first phase of breadth-first partitioning is deferred until the procedure containing the cut would otherwise succeed.

The second requirement can be viewed as a distortion of the depth limit to include the entirety of the subtree pertaining to the relation containing the cut. This approach is illustrated in Figure 4.7.

The support for cut in PrologPF is limited to the simple suspension of oracle processing while procedures containing cut are executed. The programmer is responsible for ensuring those procedures are semi-deterministic, i.e have

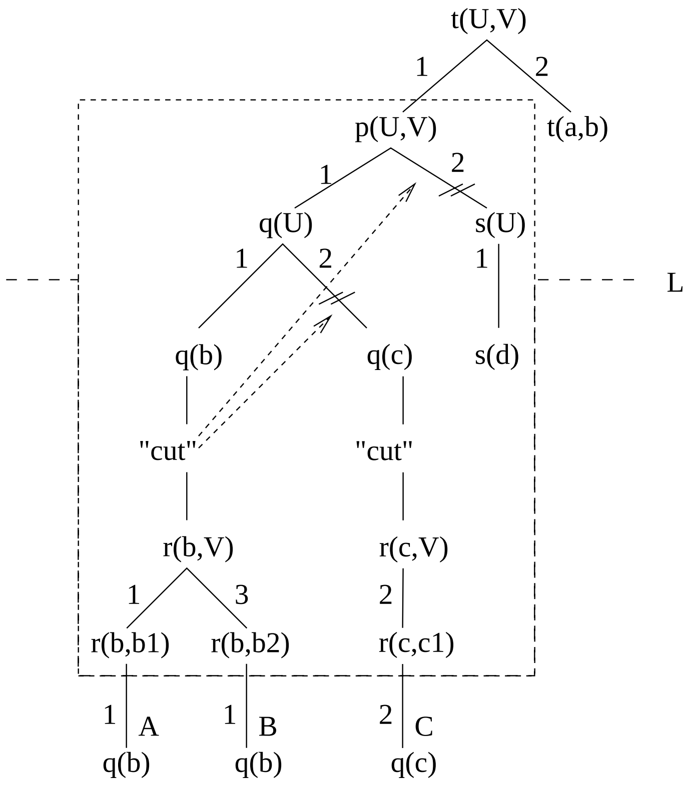


Figure 4.7: Modified depth limit definition to permit non-deterministic relations with *cut*.

only one solution or *fail*. No parallel speedup is available to procedures containing cut, and the functional support is provided to render the deterministic execution of those algorithms explicit.

4.5 Conclusions

The simple distributed implementation of the *cut* extra-logical relation would require the communication of the discovery of the cut to all path processors searching affected subtrees. Oracles can be used to identify the subtrees affected by the cut, but the communication requirements to support the distribution of the pruning information could be substantial. This distributed implementation is unsuited to the target architecture for PrologPF, in which many general purpose processors are loosely coupled with a local- or wide-area network.

Cuts can be accommodated within the distributed processing framework of PrologPF by modifying the oracle management algorithms and depth limit processing within procedures containing cut. The PrologPF compiler recognises cut within user procedures, and disables oracle processing during the execution of those procedures. The techniques discussed do not permit parallel speedup of procedures containing cut.

PrologPF aims to minimise the requirement for cut by providing support for higher-order functional programming, discussed in detail in the next chapter.

4.6 Summary

Standard Prolog [35] provides a extra-logical predicate "cut" (`!/0`) which always succeeds, but has the side effect of removing following alternative subtrees from the problem search tree. The semantics of cut are dependent upon the depth-first, left-to-right execution semantics of sequential Prolog.

Procedures containing the special predicate "cut" are often designed to be used within a particular *mode*, in which some arguments are expected to be instantiated at the time of the call while other are expected to contain logical variables. The use of procedure with a different pattern of instantiated and uninstantiated arguments can lead to unintended results. The use of cut within Prolog programs is a common source of error.

The benchmark programs used in the analysis of the distributed performance of PrologPF contain some deterministic procedures with cuts. Run-time analysis on a single cpu showed that the cuts were encountered in those sample programs between 400 and 18000 times each second. The simple implementation on multiple cpu's would be expected to increase that rate.

Oracles could be used to support a distributed implementation of the cut predicate, in which the subtrees to be pruned are identified with the associated oracle to be propagated to the affected path processors.

The target architecture for the distributed execution of PrologPF programs has a relatively high cost of communication, in contrast to an abundance of processing power within each distributed path processor. For this reason, simple support for cut is provided in which oracle processing is suspended during execution of procedures containing cut. This prevents the discovery of open oracles within any procedure containing cut. No oracles referring to a subtree within the search tree of a procedure containing cut can be generated by PrologPF, and so cannot be allocated for processing in a separate path processor.

In PrologPF, procedures containing cut must be deterministic to avoid ambiguity of the oracles skipping over the subtrees of those procedures. A modification to the definition of the depth limit used in the breadth-first partitioning strategy of PrologPF would remove that limitation.

The use of the predicate "cut" in PrologPF programs can often be avoided through the use of higher-order functional programming, a technique not available to programmers of standard Prolog. This approach forms the subject of the following chapter.

Chapter 5

Higher-Order Functions in PrologPF

5.1 Introduction

The earlier implementations of parallel Prolog exploiting the Delphi principle, described in [28, 25, 76, 49, 66], can support programs written in a pure subset of Prolog. The use of the extra-logical predicate *cut* must be avoided, as was discussed in Chapter 4.

PrologPF extends the Delphi Machine to allow the use of *cut*, but only for deterministic procedures. The programmer must avoid the intentional or accidental use of *cut* within procedures which still (in spite of the *cut*) have multiple solutions.

However, the need for *cut* within a PrologPF program is greatly reduced as support is included for the definition and application of functions, in which the deterministic execution is ensured by the system. Also, boolean functions can often be used where Prolog would rely upon the use of failure to express negation.

The higher-order functional support in PrologPF is sufficient to allow straightforward programming of all the exercises in an undergraduate ML functional programming course [60], and to allow a version of the SRI Prolog Technology Theorem Prover [70] to be implemented without *cuts*. The application of PrologPF to the functional programming exercises and PTTP is discussed in detail in Chapter 6.

PrologPF extends Prolog with support of the definition and deterministic evaluation of higher-order functions, with the functions treated as first-class values within the logic system. The Delphi oracles do not extend into the functional reduction graph, and no parallelism is provided for the evaluation of an individual function call. This is consistent with the objective of replacing Prolog procedures containing *cuts*. PrologPF does not attempt to exploit all the parallelism available in the non-deterministic but complete evaluation of functions treated as general equational theories using algorithms such as lazy narrowing. Chakravarty and Lock provide the semantics and an implementation of lazy narrowing in [20].

While PrologPF provides a consistent environment for higher-order functional programming, the language has the same syntax (with the definition of some additional operators) as normal Prolog. Thus a PrologPF program can be read by a standard Prolog compiler to produce a program in which all function applications are treated as irreducible Prolog terms.

By careful selection of the specially treated operators, the functional syntax of PrologPF will be familiar to users of Standard ML.

5.1.1 Implementation goals

1. To be compatible with the Delphi principle, functional reduction must be deterministic

2. The capabilities of the functional component of PrologPF should minimise the requirement for *cut* in the body of Prolog rules

3. The syntax should allow functional algorithms to be clearly expressed, with support for Prolog terms and variables including those representing functions, i.e. higher-order functions should be supported

4. The syntax and semantics of PrologPF should facilitate the straightforward use of functions within Prolog rules, and permit deterministic calls to Prolog procedures from within functions

5.2 Function definition: the `fun` relation

5.2.1 A PrologPF example

Before reviewing the syntax and semantics of PrologPF functions in detail with comparison to other approaches, the following examples of the factorial and append functions in PrologPF may place the alternatives in context.

Firstly, the factorial function:

```
fun fact(1) = 1;
    fact(N) = N * fact(N-1).
```

or equally (see Section 5.5.4):

```
fun fact(N) = if (N = 1)
              then 1
              else N * fact(N-1).
```

The `append` function can be defined as follows:

```
fun append(    [],Y) = Y;
    append([X|Xs],Y) = [X|append(Xs,Y)].
```

5.2.2 The PrologPF approach

Functions are defined in PrologPF with the special relation `fun/1`, which is defined as a Prolog prefix operator of low precedence with `op(1200,fx,fun)`.

Function definition in PrologPF also uses the `=` and `;` operators but the standard Prolog precedence has been maintained.

The syntax supported is shown in Table 5.1

In PrologPF, the underlying Delphi Machine has been extended to support *cut* (see Chapter 4), and this support is exploited to implement deterministic functional reduction.

Each `fun` relation is transformed through a process of *flattening* [22] into a deterministic procedure, with the actual arguments being matched against the formal parameters until a successful unification is made, at which point the choice of equality rule is committed and the reduction continuing with the term on the right-hand-side. Thus the selection of the appropriate equality rule is top-down, and the rewrite is strictly left-to-right.

The equality is required to be *constructor-based*, that is the terms in the function head must not themselves contain any defined functions. This requirement is also described as *head normal form* [42]. The syntax of the formal parameters is given in Table 5.1 as Prolog_Term, i.e. a standard Prolog term not including the application of any defined functions.

While the operational semantics of function evaluation in PrologPF have most in common with languages such as Standard ML [61, 55], the argument

Function_Definition	::=	`fun` Alternate_Definitions .
Alternate_Definitions	::=	Fun_Equality
	|	Fun_Equality ; Alternate_Definitions
Fun_Equality	::=	Fun_Head = PrologPF_Term
Fun_Head	::=	Prolog_Atom (Args...)
	|	Prolog_Atom @ [Args...]
	|	Prolog_Atom @ []
Args	::=	Prolog_Term
	|	Prolog_Term , Args
PrologPF_Term	::=	Prolog_Term
	|	Function_Application

Table 5.1: Syntax: Function Definition with the `fun` Relation

matching process is replaced with Prolog's *unification*. Argument unification in PrologPF thus differs from the matching in functional languages such as ML in two significant ways:

1. There is no requirement for left-linearity in the equality rules, i.e. variables can be repeated in the function head. The functional component of PrologPF, like the underlying Prolog, has no occurs check. As with Prolog, it is the programmer's responsibility to avoid actual parameters which would cause the unification algorithm to loop, as with the goal `:- Y = a(Y).`

2. Partially instantiated data structures (i.e. terms containing logical variables) can be passed as arguments and returned as results. This means that, for example, difference lists can be supported and that a list of variables can be appended to another.

The Prolog atom used to name a defined function denotes a function of fixed arity, set by the number of formal parameters given in the `fun` relation. Alternative definition of functions using the same name but a differing number of parameters is flagged as an error by the PrologPF compiler. This approach clearly differs from the Prolog style where a relation name can be considered a combination of the naming atom and the arity (as in `foo/2`), but is essential to permit currying within the standard Prolog syntax.

5.2.3 Alternative approaches

5.2.3.1 Deterministic relations in Prolog

Within Prolog, it is possible to define deterministic relations which then can be treated as functions:

```
fact(1,1).
fact(N,F) :- N > 1, N1 is N - 1, fact(N1,F1), F is N * F1.
```

In general, however, determinism inference is an undecidable problem, at least dependent upon the solution of the halting problem:

```
foo(X,Y) :- complicated(X,Y).
foo(X,X).
```

`foo/2` can have more than one solution only if `complicated/2` can succeed.

In many cases, the programmer uses *cut* within the Prolog program to ensure determinacy of an otherwise non-deterministic relation. For example:

```
fact(1,1) :- !.
fact(N,F) :- N1 is N - 1, fact(N1,F1), F is N * F1.
```

However, the presence of *cut* is not enough to guarantee determinacy, as in the following example:

```
a(a).
a(b) :- !.
a(c).
```

The query `:-a(X).` has the multiple solutions `X=a, X=b`.

Deterministic reduction is essential for the successful support of functions on the Delphi Machine (see Chapter 4), so the use of un-annotated Prolog relations to define functions would introduce a significant possibility of error.

5.2.3.2 Mercury

In the Mercury system, each procedure is annotated with determinism information [43]. The syntax of Prolog relation definition permits the use of relations and functions in multiple *modes*, i.e. differing arguments being instantiated at the time of the call, with others expected as results. Mercury functions are thus annotated with determinism information for each mode. For example:

```
:- pred factorial(int, int).
:- mode factorial(in,out) is det.

factorial(N, F) :-
    ( N =< 0 ->
        F = 1
    ;
        N1 is N - 1,
        factorial(N1, F1),
        F is F1 * N
    ).
```

Note that the mode information defines `factorial` to be `det`, i.e. deterministic, while the relational style of definition is retained. The Mercury compiler checks the supplied determinism information by analysis of the code. In this example the alternative representation of the function shown below would be inferred to be non-deterministic through limitations in the compiler's analysis of mutually exclusive conditions, so the earlier if-then-else form must be used:

```
factorial(0, 1).
factorial(N, F) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, F1),
    F is F1 * N.
```

The use of Mercury's determinism and type inferencing techniques have potential for exploitation on the Delphi Machine. In PrologPF all functions are, to use Mercury terminology, semi-deterministic. That is they can succeed once or fail. The issue of function failure in PrologPF is discussed in Section 5.7. Non-deterministic modes of functions are not required, and the syntax of function definition and application can be considerably simplified and optimised for the deterministic use.

5.2.3.3 Curry

The logic capabilities of the language Curry [42] are provided through the support for *non-deterministic functions*, and the function definition syntax supports this:

```
f :: Int -> Int
f 1 = 10
f 2 = 20
f 2 = 30
```

The language is typed, with `f` defined as $int \rightarrow int$ above. The call `f 2` will produce the multiple results `20` and `30`. The left-hand-sides of the functional equality definitions can be defined with conditional guards, such that the definitions are referred to as *conditional equations* where the conditions are constraints which must be solved in order for the equation to be applied. This form is used in the definition of `factorial`:

```
factorial :: Int -> Int
factorial 1 = 1
factorial n | n > 1 = n * factorial (n - 1)
```

The constraint `n` $>$ `1` is added to the second equality defining the factorial function to ensure deterministic evaluation of `factorial 1` which would otherwise match the right-hand side of both rules. To ensure deterministic execution of a function in Curry, the defining equations must be checked to ensure that the conditions are not simultaneously satisfiable [56], and no new variables can be introduced in the equations' right-hand sides.

The condition constraint in Curry can also be a boolean function expression, as an abbreviation for the rule `<bool_expr>=True`. This is similar to the treatment of function applications in relation positions in PrologPF, discussed in Section 5.6.

5.2.3.4 External procedures

The functions can be defined in a language other than Prolog, and called as external procedures. Many existing implementations of Prolog support this capability, and effort has been made to formalise the approach [14, 53, 13]. These systems do not support higher-order programming.

5.2.3.5 Logic programming with equality

A more general solution is to define functions in terms of a set of equalities [41, 57], extending Prolog's '=' relation, with conditional support provided in the form of *guards*. For example:

```
fact(1) = 1.
fact(N) = N * fact(N-1) :- N \== 1.
```

The use of guards (in this example `N \== 1`) provides access to Prolog relations, including those with multiple solutions. The use of the equality relation itself imposes no constraints on the form of the definition, permitting for example

```
append(X,append(Y,Z)) = append(append(X,Y),Z).
```

This is useful if a most general equation solving procedure is to be used, with non-deterministic selection of rewrite rules and of terms for reduction, and right-to-left as well as left-to-right application of each equality rule.

The non-deterministic solution of equations would provide interesting opportunities for the application of the Delphi principle to the extended proof tree. However, the research in this dissertation ensures the functional reduction process is deterministic such that the parallelised program has the efficiency associated with direct execution of compiled machine code.

5.3 Function application: the @ operator

The development of the `@` operator as a relation denoting function application in Prolog, with an interpretation expressed in Prolog, can be found in [27].

5.3.1 Extending Prolog for explicit function application

The standard syntax for Prolog terms is supported, with special meaning applied to a new operator `@` (defined in PrologPF as `op(600,yfx,@)`). The presence of the operator in a PrologPF term indicates that the normal unification step should be preceded by functional evaluation.

For example, in the goal for the relation "`=`":

```
:- Z = foo @ [a].
```

the term `foo @ [a]` should be evaluated before the terms `Z` and the result of `foo @ [a]` are unified with the arguments of the `=` relation.

If `foo` is a *defined function* (i.e. defined with the `fun` relation described in Section 5.2), then the rewrite rules specified in the associated `fun` relation are used for the reduction. Otherwise `foo` is a *constructor* and the term is irreducible.

For nested `@` terms, function evaluation is *strict*, i.e. innermost arguments are evaluated first. For example in:

```
:- Z = foo @ [goo @ [a], hoo @ [b]].
```

the terms `goo @ [a]` and `hoo @ [b]` will be evaluated before the results are used in the evaluation of `foo` with those arguments. The evaluation of argument terms takes place left-to-right. Evaluation ordering is significant in PrologPF because the usual functional programming one-way *matching* is replaced with *unification*, and variable arguments are permitted. The full `@` syntax is given in Table 5.2.

Function_Application	::=	Function_Term `@` `[` Args... `]`
	|	Function_Term `@` `[]`
	|	Defined_Atom `(` Args... `)`
Function_Term	::=	Defined_Atom
	|	Variable
	|	Lambda_Expression
	|	Function_Application
Lambda_Expression	::=	`lambda([` Formal_Args... `]` `,` PrologPF_Term `)`
	|	`lambda([],` PrologPF_Term `)`
Formal_Args...	::=	Prolog_Term
	|	Prolog_Term `,` Formal_Args...
Args...	::=	PrologPF_Term
	|	PrologPF_Term `,` Args...
Defined_Atom	::=	Prolog_Atom defined in earlier `fun` clause

Table 5.2: Syntax: Function Application with the `@` Operator

Note that a function is always applied to a **list** of arguments, so terms such as `foo @ a` or `foo @ X` do **not** denote function application (the correct syntax would be `foo @ [a]` and `foo @ [X]`).

A function `foo` can be defined with no arguments, and the reduction of that function can be made explicit with `foo @ []`. This use of *nil* is similar to the value *unit* in Standard ML, and is useful where function abstractions are used to emulate laziness, as in the example with infinite lists in Chapter 6. Nil argument functions are discussed further in Section 5.8.

5.3.2 Function application: syntactic sugaring

It should be noted that in PrologPF the term:

```
foo(a,b)
```

in which `foo` is a defined function, is semantically equivalent to:

```
foo @ [a,b]
```

This allows the most convenient syntax for function application to be used within PrologPF programs and allows consistent treatment of constructors and functions. For example, the solution of the goal:

```
:- Z = foo(goo(a),hoo(b)).
```

can involve functional reduction of any of `foo`, `goo`, or `hoo`. With `fun goo(X) = gg.` and `fun hoo(X) = hh.` then the goal will succeed with the single solution `Z = foo(gg,hh)`.

This consistent treatment of constructors and functions can be seen in the definition of a `wrap` function which maps a list to a similar list with each element wrapped with the constructor `envelope`:

```
fun wrap([])    = [];
    wrap([X|T]) = [envelope(X)|wrap(T)].
```

5.4 Higher-order functions and currying

A goal of the PrologPF system is to support functions as first-class data items in the extended Prolog semantics, and to permit a syntax which facilitates the straightforward creation and application of function closures.

The approach in PrologPF owes much to Standard ML [55], with support for nameless functions as lambda-expressions and the creation of closures via currying [33, 67].

5.4.1 Lambda-expressions

Nameless functions are created in PrologPF using the special constructor `lambda/2`. The syntax is given in Table 5.2.

An example of a goal using a lambda expression representing the increment function is:

```
:- Z = lambda([X],X+1) @ [6].
```

returning the single solution `Z = 7`.

As with defined functions in PrologPF, the evaluation of the function term proceeds with the unification of the actual parameter (in this example `6`) with the argument of the lambda expression (`X`). The instantiated second argument of the `lambda` term is then evaluated to produce the final result.

Unlike standard Prolog, the scope of the formal arguments of the lambda expression (`X` in the example above) is limited to that expression. This ensures the correct operation of goals such as:

```
    :- Y = lambda([X],X+1) @ [6], Z = lambda([X],X*2) @ [7].
```

PrologPF lambda terms can be defined to take **no** arguments, providing a mechanism to delay the evaluation of the expression given as the second argument. For example:

```
                              Z = lambda([],f(100))
```

The expression `f(100)` will not be evaluated until a subsequent application `Z @ []`. This use of *nil* arguments is discussed further in Section 5.8.

5.4.2 Currying

The support for currying in PrologPF ensures that the following equivalence holds true:

$$\texttt{foo @ [a] @ [b] @ [c]} \equiv \texttt{foo @ [a,b,c]}$$

The arity of a defined function is fixed in the `fun` relation (Section 5.2). Any alternate definition using the same function name but with a differing number of formal parameters is flagged by PrologPF as an error. This means the PrologPF compiler can generate appropriate code to return a lambda expression where a function is called with fewer arguments than appear in the `fun` definition. The definition of the operator `@` was shown in Section 5.3 to be left-associative (the 'yfx' in `op(600,yfx,@)`.

These capabilities combine to provide the flexible support for higher-order abstraction through the partial application of functions, known as currying.

For example, if a function `foo` is defined with 3 arguments as in:
`fun foo(X,Y,Z) = X+Y+Z.`
then (using symbol $\rightsquigarrow$ to represent 'evaluates to'):
`foo @ [a]` $\rightsquigarrow$ `lambda([Y,Z],foo(a,Y,Z))`
$\Longrightarrow$

```
 foo @ [a] @ [b] @ [c]  ≡   ((foo @ [a]) @ [b]) @ [c]
                        ⇝   (lambda([Y,Z],foo(a,Y,Z)) @ [b]) @ [c]
                        ⇝   lambda([Z],foo(a,b,Z)) @ [c]
                        ⇝   foo(a,b,c)
                        ≡   foo @ [a,b,c]
```

The explicit use of the `@` operator and the use of currying permit the straightforward definition and application of functions such a `map`:

```
fun map(F,[]) = [];                            % map definition
    map(F,[X|Xs]) = [F @ [X]|map(F,Xs)].

:- Z = map(+1,[10,20,30]).                     % curried +
:- Inc = map(+1), Z = Inc @ [[10,20,30]].      % curried map, +
```

Each query succeeds with the single solution for `Z = [11,21,31]`.

5.5 Special treatment of `if-then-else`

PrologPF includes a predefined function `if` to provide conditional evaluation of alternative expressions. The systematic eager evaluation in PrologPF precludes the definition of `if` as a normal PrologPF function with three arguments:

```
fun if(true, A,B) = A;
    if(false,A,B) = B.
```

As the argument evaluation semantics of PrologPF are eager, in an expression such as `if(Z=0, 1, 100/Z)` all three arguments would be evaluated before the application of `if`, producing a possible run-time arithmetic error during the attempted evaluation of `100/Z`.

To provide more useful behaviour, `if` is treated as a predefined function with exceptional semantics. The special treatment is unique to `if`:

1. The evaluation of the alternative expressions is delayed until **after** the condition has determined which of the two alternatives should be evaluated. Only **one** of the two alternatives will then be evaluated.

If_Expression	::=	`if(`PrologPF_Term$_1$`,` PrologPF_Term$_2$`,` PrologPF_Term$_3$`)`
	|	`if` PrologPF_Term$_1$ `then` PrologPF_Term$_2$ `else` PrologPF_Term$_3$
	|	`if` PrologPF_Term$_1$ `then` PrologPF_Term$_2$

Table 5.3: Syntax: `if`

2. The condition term is treated as a Prolog **goal**, rather than a boolean-valued reducible expression

5.5.1 Syntax

The syntax for the conditional `if` expression is given in Table 5.3.

The use of the predefined operators `if`, `then` and `else` is permitted to reduce the use of brackets and allow a syntax similar to that of languages such as Standard ML. Where the `if-then-else` form is used, the resultant expression is equivalent to the term `if(Term`$_1$`,Term`$_2$`,Term`$_3$`)`.

To allow a convenient syntax without modifying the precedence of the standard Prolog operators, the following precedences are used for `if`, `then` and `else`:

```
:- op(675,fx,if).      % 'if' is prefix
:- op(650,xfx,then).   % 'then' is infix
:- op(625,xfx,else).   % 'else' is infix
```

The precedence of the predefined `if`, `then` and `else` operators in PrologPF implies that:
`if` Term$_1$ `then` Term$_2$ `else` Term$_3$
$\equiv$ `if (`Term$_1$ `then (`Term$_2$ `else` Term$_3$ `))`
$\equiv$ `if(then(`Term$_1$,`else(`Term$_2$,Term$_3$`)))`

The `else`-expression can be omitted, such that:
`if` Term$_1$ `then` Term$_2$ $\equiv$ `if` Term$_1$ `then` Term$_2$ `else fail`

The precedence of the `if-then-else` compound term has been set higher than that of the Prolog's `=` and `;` operators to minimise the need for brackets in function definitions, and in goals of the form `Z = if-expression`. The compromise means that conditional operators used in `if` conditions (i.e. Term$_1$) must be bracketed, as must be nested `if` expressions.

For example:

```
if (A < 20) then (if (A > 12)
                   then middle
                   else lower
                  )
            else upper
```

5.5.2 Evaluation

Special code is generated in the call to `if` in the evaluation of if-expressions.

5.5.2.1 Defined evaluation ordering with `if`

For any other arity/3 function call such as `foo(`Term$_1$`,`Term$_2$`,`Term$_3$`)` for defined function `foo`, code of the following form would be generated:

[code to evaluate Term$_1$ with result as term X_1]
[code to evaluate Term$_2$ with result as term X_2]
[code to evaluate Term$_3$ with result as term X_3]
functional evaluation of `foo(`X_1,X_2,X_3`)`

In the case of the special function `if` the eager evaluation of both alternative expressions in terms such as `if (Z = 0) then 1 else 100/Z` would not execute as intended for `Z = 0`, so consequently code of the following form will be generated:

[code to find first solution of call(Term$_1$) as relational goal] (Section 5.5.2.2)
<on success:> *[code to return result of evaluation of Term$_2$]*
<on failure:> *[code to return result of evaluation of Term$_3$]*

PrologPF ensures that:

1. The condition goal completes **before** the evaluation of the alternate expressions of the `if`-expression.
2. The condition goal succeeds with one solution, or fails.
3. Only one of the alternate expressions will be evaluated: the `then` expression if the condition goal succeeds, or the `else` expression if it fails.

5.5.2.2 `if` condition as relational goal

There is considerable advantage in giving functions within the combined functional logic system access to the relations in the program and those in the Prolog libraries. The implementation chosen for the Delphi Machine requires that the function evaluation be deterministic. A successful compromise has been achieved with:

1. The **only** place a function in PrologPF can call a Prolog relation is in the condition of an `if`-expression
2. The call uses Prolog's normal search, but determinism is maintained with first-solution semantics
3. The acceptance of boolean functions as relational goals reintroduces functional terms as conditions (Section 5.6)

An example showing how the Prolog library `append` relation can be used to produce a similar (but deterministic) function would be:

```
fun append(X,Y) = if append(X,Y,Z) then Z.
```

This example relies upon the following:

1. The `if` semantics ensure the goal `append` produces a value for `Z` before the evaluation of the sub-expressions `Z` and `fail`.
2. The `if` semantics ensure that only one of the sub-expressions is evaluated, after the solution of the conditional goal.
3. The relation `append/3` and the function `append/2` are recognised as having different names (see Section 5.6)
4. The missing `else`-expression is equivalent to `else fail`, so the definition is an abbreviation for:
   ```
         fun append(X,Y) = if append(X,Y,Z) then Z else fail.
   ```
5. The predefined function `fail` is available to produce function failure (Section 5.7)

The use of relational goals as conditions, combined with Prolog's left-to-right search rule, leads to a Prolog syntax with semantics similar to the special operators in languages such as Standard ML for `andalso` and `orelse` [55]:

Conjunction:	`(P,Q,R)`	$\equiv$	`P andalso Q andalso R`
Disjunction:	`(P;Q;R)`	$\equiv$	`P orelse Q orelse R`

For example, using the standard Prolog library relations > and <:

```
fun account_status(Bal) = if (Bal > 0, Bal < 100)
                          then normal
                          else needs_attention.
```

In using a relational goal as the condition, the PrologPF `if` expression has similar behaviour to the Prolog conditional goal, written `A -> B; C.` The definition of the operators "`->`" and "`;`" are provide in [35]. The subgoal `A` is called to provide one solution or fail. In the former case, subgoal `B` is then called, else subgoal `C` is called. The semantics are complicated by the presence of any cuts in subgoals `A`, `B` or `C`. The deterministic execution of functions in PrologPF permits the provision of an *if-then-else* expression without these complexities.

5.5.3 Value declarations

A value declaration gives an expression a *name* within a particular *scope*.

The PrologPF support for `if` if ensures that the relational condition is executed before the alternate expressions. The unifier of the free variables in the condition is thus valid for the evaluation of the `then`-expression, which is only evaluated if the condition has succeeded. Thus the use of the `=` relation in the condition of an `if-then-else` expression can give a value a name, which will be valid in the scope of the `then` sub-expression.

The use of the unification of the condition to support naming in this way is convenient if a sub-expression is to be repeated within an expression, as often occurs within an `if-then-else`. An example is in a definition of a `max` function to find the highest integer in a list:

```
fun max([X])     = X;
    max([X|Xs]) = if (M = max(Xs))
                  then (if (X > M)
                        then X
                        else M
                       ).
```

In the recursive case, the condition goal `M = max(Xs)` results in the evaluation of `max(Xs)` being unified with a new free variable `M`, with the unifier `M/n` (where n is the largest integer in `Xs`) being valid for the subsequent evaluation of `if (X > M) then X else M`.

The use of M as a *name* to represent the value max(Xs) is equivalent to the repeated appearance of the value in the then expression. The max function could equally be written:

```
fun max([X])    = X;
    max([X|Xs]) = if (X > max(Xs))
                  then X
                  else max(Xs).
```

As these value declarations are using the standard = relation in the condition, the method supports a convenient technique for using functions that return multiple results as a tuple. This can be seen with the second of the complementary functions zip and unzip. The function zip takes two lists of equal length as arguments, and returns a list of pairs [42]:

```
fun zip([],[])         = [];
    zip([X|Xs],[Y|Ys]) = [(X,Y)|zip(Xs,Ys)].
```

The complementary function unzip has a convenient definition using a value declaration [61]:

```
fun unzip([])             = ([],[]);
    unzip([(X,Y)|Pairs]) = if ((Xs,Ys) = unzip(Pairs))
                            then ([X|Xs],[Y|Ys]).
```

A version of unzip that did not use a value declaration could be written using of auxiliary functions to extract the elements of the tuple and repeating the unzip(Pairs) sub-expression. Alternatively, an auxiliary function could be defined to add a pair of elements to pair of lists, as in:

```
fun addpair((X,Y),(Xs,Ys)) = ([X|Xs],[Y|Ys]).

fun unzip([])           = ([],[]);
    unzip([Pair|Pairs]) = addpair(Pair, unzip(Pairs)).
```

However, the use of value declarations in unzip avoids the use of auxiliary functions.

In the absence of common-expression elimination optimisations in the PrologPF compiler, the use of value declarations results in a more efficient object program. In general, the use of names to represent expressions that are complex or repeated can result in programs that are more comprehensible.

5.5.4 Alternate function definitions ≡ `if`

In PrologPF, the function definition style using alternate argument patterns can be shown to be equivalent to a single functional equality using the predefined `if` function.

The following characteristics of the `if` function are important in the equivalence:

- The defined lazy conditional evaluation of the arguments to `if`
- The call to the condition goal is defined to *precede* the evaluation of one of the functional terms. Value declarations from unification of terms in the condition goal with local variables are therefore guaranteed to be bound in the scope of the subsequently evaluated dependent expression.

With the example of the factorial function:

```
fun fact(1) = 1;
    fact(N) = N * fact(N-1).
```

The equivalent if-then-else form is:

```
fun fact(Z) = if (Z=1)
              then 1
              else (if (Z=N)
                    then N * fact(N-1)
                   ).
```

The general form of the translation is:

```
fun Function_name(Arg_pattern1, Arg_pattern2...) = Expression_1;
    Function_name(Arg_pattern3, Arg_pattern4...) = Expression_2...
```

goes to:

```
fun Function_name(Var1,Var2...)
          = if (Var1 = Arg_pattern1, Var2 = Arg_pattern2...)
            then Expression_1
            else (if (Var1 = Arg_pattern3, Var2 = Arg_pattern4...)
                  then Expression_2
                  else ...
                 ).
```

Value declarations in the relational goal of the `if` condition can be seen more clearly with the transformation of the `append` function:

```
fun append(     [],Y) = Y;
    append([X|Xs],Y) = [X|append(Xs,Y)].
```

goes to:

```
fun append(L,Y) = if (L=[])
                  then Y
                  else (if (L=[X|Xs])
                        then [X|append(Xs,Y)]
                       ).
```

5.5.5 Summary of PrologPF `if` semantics

The goal of the `if-then-else` implementation in PrologPF is to provide useful conditional evaluation semantics, while supporting deterministic access to relations.

With the expression `if` Term_1 `then` Term_2 `else` Term_3:

- The conditional expression Term_1 is treated as a relational goal, either succeeding with a variable binding, or failing.
- The depth-first, left-to-right search of standard Prolog is used to find a solution to Term_1, and the search is limited to finding the first solution.
- The call to the conditional expression Term_1 completes before the evaluation of either Term_2 or Term_3.
- If Term_1 succeeds then Term_2 is evaluated in the context of any bindings resulting from the solution of Term_1, and the result returned as the value of the `if`-expression.
- If Term_1 fails then Term_3 is evaluated and returned as the result of the `if`-expression.
- If the `else`-expression (`else` Term_3) is omitted, the semantics are the same as if an `else`-expression (`else fail`) were added.

5.6 Boolean functions as relations

In summary, the following equivalence holds for functions used in the position of relational goals:

`?- foo(a).` $\equiv$ `?- foo(a) = true`

iff `foo` is a defined function of arity/1.

A function application term is permitted to appear in the position of a relational goal, where it is treated as a call to the Prolog `=` relation to unify the result of the function application with `true`. This applies to the body of each rule and the condition of each `if` expression.

For example, with a boolean function `prime(X)` returning true for a prime argument and false otherwise, the goal:

```
?- p(X), prime(X), write(X).
```

is equivalent to:

```
?- p(X), prime(X) = true, write(X).
```

The explicit treatment of boolean functions as relations in this way can be seen in the prototype produced by Paulson and Smith [62]. The language Escher [51] has **all** relations declared as boolean functions in this way.

Either the explicit `@` operator can be used to denote the function application, or the Prolog compound term syntax can be used. In the latter syntax, the outermost functor of the goal will only be recognised as a defined function if the number of actual parameters matches the arity of the defined function of the same name. The specification of a reduced number of arguments in a curried application is not useful where a boolean result is required. A partial (i.e. curried) function application would always return a higher-order result, such that:

(`<curried_application> = true`) $\equiv$ `fail`

The requirement for the arities of the defined function and the actual use within a goal facilitates the conversion of library relations (such as `append`) into functions and vice versa. I.e. the functional definition of `append/2` does not conflict with the relational definition `append/3`, and the library relation can be used in the function definition:

```
fun append(X,Y) = if append(X,Y,Z) then Z.
```

Equally, the deterministic functional version of append given in Section 5.5.4 could have been used for a version of the library relation limited to deterministic modes:

```
append(X,Y,Z) :- Z = append(X,Y).
```

To summarise the naming/arity issues arising from both currying and the acceptance of boolean functions as relations:

1. Each alternate equality statement in the definition of a function must have the same number of formal parameters, and this number is the arity of the function.
2. A function can have the same name as a relation, but must not have the same arity.

The first rule is to allow currying, the second to allow boolean functions as goals. The functional logic language Mercury has a similar rule to 2 above, but in Mercury a function must not have an arity that is **one less** that a relation of the same name. The Mercury name/arity constraints are inconvenient, as it is natural to define a function (such as `append/2`) to have an arity one less that an equivalent relation (i.e. `append/3`). PrologPF exploits this capability to define functions representing deterministic modes of many frequently-used library relations such as `append` and `=..`.

In the design of PrologPF, a choice was made to introduce rule 2, rather than the alternative that boolean functions as goals should require explicit use of the `@` operator. The body of Prolog code converted for execution on the PrologPF system has not yet included enough examples of relations with multiple arities to confirm this design decision.

5.7 Failure of functions

The functional support in PrologPF is embedded within an environment of relations which are expected to `succeed` (with an associated variable binding) or *fail*. The treatment of function applications as relation argument terms associates every application with an underlying relation, for example in:

```
?- Z = fact(5).
```

the function application of `fact` is as an argument of the relation `=`.

5.7.1 Functional failure ⇒ Relation failure

In PrologPF, function failure is supported through the provision of a special term `fail`. This mirrors the standard Prolog relation `fail`, which always fails.

1. The evaluation of the term `fail` within an expression produces no value but always fails.

2. A function application fails if evaluation of a subexpression in the right-hand-side of the associated definition fails.

3. A relation fails if the evaluation of a functional argument fails.

The use of `fail` within a function definition can be seen in the `lookup` function, which returns a value associated with a key in a list of paired key-value terms:

```
fun lookup(_,[])                  = fail;
    lookup(Key,[(Key,Value)|_]) = Value;
    lookup(Key,[_|T])           = lookup(Key,T).
```

The function might be used in a program such as:

```
a(a).
a(b).
a(c).

?- a(X), write(lookup(X,[(a,1),(c,3),(e,5)])).
```

The subgoal `a(X)` produces values `a`, `b` and `c` for `X`, calling `write` with the value of the `lookup` application. As the key-value list argument contains no entry for `b`, the application will fail for that argument value. Backtracking will take place as in standard Prolog, such that `write` will display the values `1` and `3` from the successful application of `lookup` with `a` and `c`.

5.7.2 Function `fail` as an exception

Within the function evaluation, the semantics of `fail` are those of an uncaught *exception*. An introduction to exceptions in Standard ML can be found in [61]. In PrologPF, the exception can be considered to be caught at the point immediately preceding the unification of the term with the corresponding argument of the relation, where it causes that relation to fail.

The general support for exceptions would be consistent with the rest of the functional support in PrologPF as

1. Function evaluation in PrologPF is innermost nested term first (referred to as *eager*), so the evaluation of the expression term to be raised as an exception can occur **before** the exception is raised and the **value** of the expression returned as the exception value. A lazy functional language with *call-by-need* argument evaluation semantics

would require special treatment of the expressions given to the `throw` function.

2. PrologPF permits partially defined functions (where some legal actual argument patterns have no matching left-hand-side in the function definition) and function failure. A more general support for exceptions can be provided for which these are special cases.

If, as in Standard ML [55], a general support for exceptions were provided though the use of `raise` and `handle` operators, then the use of `fail` within PrologPF could be shown to be equivalent to the limited use of those exceptions:

`fail` in PrologPF	$\equiv$	`raise Fail` with declared `exception Fail`
With relation R, argument expressions $e_1, ..., e_n$, and goal $R(e_1, \ldots, e_n)$	$\equiv$	e_i `handle Fail` $\Rightarrow$ ensure *failure* of R at each argument $e_i, i = 1 \ldots n$

5.7.2.1 A proposal for more general exception support in PrologPF

Standard Prolog [35] has support for exceptions at the level of relations with the predefined `catch` and `throw` meta-logical operators. An exception is generally referred to as a `Ball`.

The format for the use of `throw` is:

```
throw(Ball)
```

where `Ball` is any Prolog term to be propagated as an exception. Similarly the format for the use of `catch` is:

```
catch(Goal,Ball,Handler)
```

where:

`Goal` is a Prolog relational goal potentially containing `throw` subgoals

`Ball` is a term to be unified with the actual argument of any `throw` operators encountered during execution of `Goal`

`Handler` is a subgoal to be called when an exception is caught, i.e. successfully unified with `Ball`

Often, `Ball` and `Handler` will contain common free variables, as a means of propagating values from the `throw`.

The goal:

```
catch(throw(foo),X,write(X))
```

will have the effect of writing "foo" to the output, with the execution proceeding as follows:

1. `catch` calls the subgoal given as its first argument, namely `throw(foo)`.
2. The subgoal `throw(foo)` throws (raises) the ball (exception) `foo`.
3. The ball `foo` propagates to the level of the surrounding `catch` where it is unified with the second argument of the `catch` relation (`X`). If this unification had failed, then the ball continues to propagate to any higher enclosing `catch` relation.
4. With the successful unification of `foo` with `X`, the subgoal `write(X)` given as the third argument to `catch` is called.
5. `foo` is written to the output.

In the context of standard Prolog's `catch` and `throw`, the use of `fail` within defined functions in PrologPF can be treated as:

`fail` in PrologPF	$\equiv$	`if throw(fail) then _ else _`
A goal containing relation R, as in $\ldots, R, \ldots$	$\equiv$	$\ldots$,`catch(`R`,fail,fail)`,$\ldots$

Note the use of `if-then-else` to map the relational call to `throw` into an expression. The implicit `catch` which can be considered to be wrapped around each relation call containing functional arguments is shown to only handle one value of exception (`fail`). The `catch` goal will then fail if this type of exception is caught.

With this definition we arrive at the semantics for our use of `fail` within functions as uncaught exceptions, leading to failure of the associated relation.

The definition using `catch` and `throw` could lead to the more flexible use of exceptions within the functional component of PrologPF, although the implementation to date only permits the support for `fail`.

An improved support would:

- Allow any term to be raised as an exception value within a defined function, for example `throw(foo)`.

- Allow exceptions to be caught within the functions rather than propagating to the relational level.
- Treat any uncaught exception from a functional evaluation as `fail`.

The implementation would require the following:

- The meta-relation `throw` should be mapped to a similar function `throw/1`, where an expression `throw(X)` would have the same meaning as
  `if throw(exception(X)) then _ else _.`
  The definition would use the support in PrologPF for relations as if-conditions. The use of anonymous variables as the alternate expressions is arbitrary, as the function `throw` would never return any value. Function could then use `throw` within any expression.
- As with the meta-relation `catch`, a functional equivalent would allow the handling of exceptions at any level of a nested functional expression, as in Standard ML. The ML syntax is
  E `handle` $P_1 => E_1| \ldots |P_n => E_n$
  where E is the expression which may possibly raise an exception, P_i is an expression matching the exception and E_i is the corresponding value to be returned instead of E. The equivalent support in PrologPF would be by nested applications of a `catch` function, which would have the same capabilities as `catch(`E`,exception(`P_i`),`E_i`)` for each pattern P_i for unification with the exception term thrown.
- The implicit `catch` wrapper around each relation R would be
  `catch(`R`,exception(X),fail).`
  This can be contrasted with the more limited form supporting `fail` given above.

5.8 Unit

ML has a built-in type '`unit`' with only one member, namely "`()`". A function of intended arity zero will be defined of type "`unit` $\rightarrow \alpha$", and the value of that function will be returned by the explicit application of that function to "`()`".

An example ML function definition of this type is:

```
>fun foo () = 22;
foo: unit -> int
>val a = foo ();
a = 22 : int
```

In PrologPF, all functions are explicitly applied to a **list** of actual arguments, using Prolog syntax for lists, and the application of a function to **no** arguments can be explicit by using an empty list (i.e. nil: "`[]`"). The application of a function to no arguments simply returns that function, i.e.
`foo @ []` for defined function `foo` with arity $0 \equiv$ evaluated `foo`
`bah @ []` for arity `bah` > 0 is $\equiv$ `bah`
$\Rightarrow$ `bah @ [] @ [] @ []` $\equiv$ `bah`
$\Rightarrow$ `bah @ [] @ [] @ [X]` $\equiv$ `bah @ [X]`.

5.9 The interaction of functions and relations

In the combined functional and logic programming paradigm of PrologPF, most effort has been placed in the design of the overlap between the use of defined functions and relational rules. The resultant system allows the exploitation of defined functions within rules and access to relations from within functions in a straightforward way with clear semantics.

The interaction between the functional and logic elements of a PrologPF program is limited to:

Function definition. The relation `fun` is given special meaning as declaring the ordered equational rewrite rules defining a named function.

Function application. The semantics of the actual argument terms of predicates has been extended to include the application of defined functions with the special operator `@`. The functional reduction is defined to occur as a step preceding the unification of the resultant term with the predicate formal arguments.

Function failure. Function failure is defined, such that a goal with a failing function as an argument term is defined to fail.

Relation call from within functions. The condition term of the built-in PrologPF function `if` is defined to be a relational goal, with determinism ensured by one-solution call semantics.

Functions as goals. The non-curried application of a defined function as a goal is defined to be equivalent to the `=` goal with that application term and `true`.

Functions as first-class data items. A function abstraction returned as the result of a higher-order function or the user definition of a lambda-term can be unified with a logical variable for application within subsequent goals or sub-goals.

5.10 Some PrologPF examples

A comprehensive review of the application of PrologPF to both logic and functional problems is given in Chapter 6.

PrologPF examples of functions for factorial, append, map, and max have been given in the preceding sections, and are repeated here for clarity:

```
fun fact(1) = 1;
    fact(N) = N * fact(N-1).

fun append(    [],Y) = Y;
    append([X|Xs],Y) = [X|append(Xs,Y)].

fun map(F,[]) = [];
    map(F,[X|Xs]) = [F @ [X]|map(F,Xs)].

fun max([X])    = X;
    max([X|Xs]) = if (M = max(Xs))
                  then (if (X > M) then X else M).
```

5.10.1 Undergraduate Prolog exercise attempt

An interesting example of functional logic syntax could be seen in an attempt by an undergraduate to write a relation `remhigh/2` in which the first argument is a list of integers, and the second is the same list excluding the highest element. The undergraduate wrote:

```
%%%% remhigh:    L2 is list L1 excluding highest member of L1

remhigh(L1,L2) :- remove( max(L1), L1, L2).

%%%% remove(Element, List, Remainder_list) :
%%%%     Remainder_list is List excluding Element

remove(N, [N|T], T).
remove(N, [H|T], [H|T1]) :- N \== H, remove(N, T, T1).
```

From the definition of `remhigh` it can be seen that the student expected a functional support that is not present in Prolog. The student is also suggesting a natural syntax. The above attempt would be correct in PrologPF with the definition of `max` given above in Section 5.5.

5.10.2 Lazy lists

This example is extended and reviewed in more detail in Chapter 6, where infinite streams of primes are created. Here we will show the use of the higher-order features of PrologPF to represent infinite lists.

Infinite lists in this program will be represented by constructor terms of the form:

```
item(Head,Tail)
```

where `Head` is the value at the head of the list and `Tail` is a *function* of arity zero which returns the tail of the list. The empty list can be represented by a constructor such as `empty`. The functions to extract the components of a list are:

```
fun head(empty)     = fail;
    head(item(X,_)) = X.

fun tail(empty)     = fail;
    tail(item(_,F)) = F@[].
```

A function to create the infinite list of natural numbers is:

```
     fun make_nats(N) = item(N,lambda([],make_nats(N+1))).
```

The application `make_nats(N)` can now be used to represent an infinite list the natural numbers starting from N.

A goal such as `?- Z = head(tail(tail(make_nats(1)))).` will return the expected solution `Z = 3`. With this representation of infinite lists, a version of the higher-order function `map` can be defined in PrologPF:

```
fun imap(F,empty)     = empty;
    imap(F,item(X,T)) = item(F@[X], lambda([],imap(F,T@[]))).
```

The function can be demonstrated in a goal such as

```
        ?- Z = head(tail(tail(imap(*2,make_nats(1))))).
```

giving the solution `Z = 6`.

The `imap` function illustrates the combined use of *constructors* (`empty`,`item`), *higher-order variables* (`F`), explicit application with `@`, implicit application of `imap`, use of `lambda` expressions, and the use of *nil* to denote evaluation of an arity/0 function. The example shows that the syntax facilitates the use of these capabilities without obscure programming constructs.

5.11 Comparison of PrologPF with `call/N`, `apply/3`

The semantics of the support for functions in PrologPF has most in common with Naish's `apply/3` [58], although he retains the definition of functions as Prolog relations, and permits non-deterministic evaluation. Naish's definition of `apply/3` is designed as a more capable replacement for the `call/N` extra-logical predicate provided in some Prologs and used as the basis for the higher-order functional support in Mercury [69].

Table 5.4 compares PrologPF with `call/N` and `apply/3` using the examples from [58].

`call/N`	`apply/3`	`PrologPF`
`map(F,[],[]).`	`map(F,[],[]).`	`fun map(F,[])     = [];`
`map(F,[X|Xs],[Y|Ys]) :-`	`map(F,[X|Xs],[Y|Ys]) :-`	`    map(F,[X|Xs]) =`
		`        [F @ [X]|map(F,Xs)]`
`    call(F,X,Y),`	`    apply(F,X,Y),`	
`    map(F,Xs,Ys).`	`    map(F,Xs,Ys).`	
`filter(P,[],[]).`	`filter(P,[],[]).`	`fun filter(P,[])     = [];`
`filter(P,[X|Xs],Ys) :-`	`filter(P,[X|Xs],Ys) :-`	`    filter(P,[X|Xs]) =`
`    (call(P,X) ->`	`    (apply(P,X,true) ->`	`        if (P @ [X])`
`       Ys = [X|Z]`	`       Ys = [X|Z]`	`        then [X|filter(P,Xs)]`
`    ;`	`    ;`	`        else filter(P,Xs).`
`       Ys = Z`	`       Ys = Z`	
`    ),`	`    )`	
`    filter(P,Xs,Z).`	`    filter(P,Xs,Z).`	
`foldr(F,B,[],B).`	`foldr(F,B,[],B).`	`fun foldr(F,B,[])     = B;`
`foldr(F,B,[X|Xs],R) :-`	`foldr(F,B,[X|Xs],R) :-`	`    foldr(F,B,[X|Xs]) =`
`    foldr(F,B,Xs,R1),`	`    foldr(F,B,Xs,R1),`	`        F @ [X,foldr(F,B,Xs)].`
`    call(F,A,R1,R).`	`    apply(F,X,FA),`	
	`    apply(FA,R1,R).`	
`compose(F,G,X,FGX) :-`	`compose(F,G,X,FGX) :-`	`fun compose(F,G,X) =`
		`        F @ [G @ [X]].`
`    call(G,X,GX),`	`    apply(G,X,GX),`	
`    call(F,GX,FGX).`	`    apply(F,GX,FGX).`	
`converse(F,X,Y,FYX) :-`	`converse(F,X,Y,FYX) :-`	`fun converse(F,X,Y) =`
		`        F @ [Y,X].`
`    call(F,Y,X,FYX).`	`    apply(F,Y,FY),`	
	`    apply(FY,X,FYX).`	

Table 5.4: Comparison of `call/N`, `apply/3` and PrologPF

The above relations and functions are then tested against the queries in Table 5.11 [58].

With the syntax shown in the right-hand column, PrologPF can support

	call/N, apply/3	PrologPF
1.	`filter(>(5),[3,4,5,6,7],As)`	`As = filter(>(5),[3,4,5,6,7])`
2.	`map(plus(1),[2,3,4],As)`	`As = map(+1,[2,3,4])`
3.	`map(between(1),[2,3],As)`	⇒ `non-deterministic function`
4.	`map(plus(1),As,[3,4,5])`	⇒ `reversible map, plus`
5.	`map(plus(X),[2,3,4],[3,4,5])`	⇒ `reversible plus`
6.	`map(plus(X),[2,A,4],[3,4,B])`	⇒ `reversible plus`
7.	`map(plus(X),[A,3,4],[3,4,B])`	⇒ `reversible plus`
8.	`foldr(append,[],[[2],[3,4],[5]],As)`	`As = foldr(append,[],[[2],[3,4],[5]])`
9.	`foldr(converse(append), [], [[2],[3,4],[5]], As ).`	`As = foldr(converse(append), [], [[2],[3,4],[5]] ).`
10.	`compose(map(plus(1)), foldr(append,[]), [[2],[3,4],[5]], As ).`	`As = map(+1) @ [foldr(append,[]) @ [[2],[3,4],[5]] ].`
11.	`foldr(compose(append,map(plus(1))), [], [[2],[3,4],[5]], As ).`	`As = foldr(compose(append, map(+1)), [], [[2],[3,4],[5]] ).`
12.	`map(plus,[2,3,4],As).`	`As = map(+,[2,3,4]).`

Table 5.5: Queries from [58] for `call/N`, `apply/3`, PrologPF

the functional examples given in [58] with the exception of those requiring multiple answers (3) or reversible functions (4-7). `Call/N` does not provide reversible functions (4-7) or permit general higher-order programming as in (11-12). `Apply/3` does not provide reversible functions (4-7). A discussion of the significant features of each example is given below (and in [Nai96]), followed here by some more examples highlighting the capabilities of PrologPF.

1. `filter(>(5),[3,4,5,6,7],As)`
   The function $>$ passed to `filter` is curried, representing the boolean function $\lambda x \rightarrow (5 > x)$. The higher-order function `filter` applies this argument to `[3,4,5,6,7]`, returning `[3,4]`. The example exercises the definition of higher-order functions and currying.

2. `map(plus(1),[2,3,4],As)`
   In a similar fashion to example 1, the curried function `plus(1)` is passed to the higher-order function `map`. In PrologPF the function and predicate name-spaces are distinct (see Section 5.6), so the plus function can be given the name `+` rather than a special relation being

needed. The PrologPF library includes the definitions of all the arithmetic functions, e.g. `fun +(X,Y) = if (Z is X+Y) then Z else fail..` The `is` relation is redundant in PrologPF.

3. `map(between(1),[2,3],As)`
   The relation `between(I,J,X)` has multiple solutions, and its call from within a functional expression in PrologPF such as
   `if between(1,9,N) then N else 0`
   would ensure deterministic execution of the predicate. This would enforce a single solution or failure. Example 3 has no equivalent in the functional component of PrologPF, as that would conflict with the implementation on the Delphi Machine.

4. `map(plus(1),As,[3,4,5])`
   Examples 4 through 7 require the functions `map` or `plus` to be reversible. None of `call/N`, `apply/3` or PrologPF provides support for reversible functions.

5. `map(plus(X),[2,3,4],[3,4,5])`
   See 4 above.

6. `map(plus(X),[2,A,4],[3,4,B])`
   See 4 above.

7. `map(plus(X),[A,3,4],[3,4,B])`
   See 4 above.

8. `foldr(append,[],[[2],[3,4],[5]],As)`
   The higher-order function `foldr` accepts a function abstraction (in this case the function `append`) and recursively applies it to the argument list, treating the argument `[]` and the final element. With `call/N` and `apply/3`, the first call to `append` is with the last element of the list of lists and `[]`, e.g. `append([5],[],R)`, where `R` is an intermediate result. Similarly, PrologPF stacks the intermediate result of `append([5],[])`. Each call to `append` is with both required arguments ground, and `call/N`, `apply/3` and PrologPF provide the flattened solution `[2,3,4,5]`.

9. `foldr(converse(append),[],[[2],[3,4],[5]],As)`
   The example proceeds in a similar manner to example 8, with the function abstraction provided by `converse(append)`. When called by `foldr`, the abstraction is passed both required arguments which are appended in reverse, resulting in the solution `[5,3,4,2]`.

10. `compose(map(plus(1)),foldr(append,[]),[[2],[3,4],[5]],As)`
    This is a more complex combination of currying and higher-order functions, but with similar system requirements to examples 8 and 9.

`map(plus(1))` increments each member of a list, while `foldr(append,[])` flattens a list of lists, so the term represents:

increment_list(flatten_list([[2],[3,4],[5]])).

This can be represented more naturally in PrologPF than in the flat syntax with `call/N` and `apply/3`.

11. `foldr(compose(append,map(plus(1))),[],[[2],[3,4],[5]],As)`
    This example is evaluated successfully with `apply/3` and in PrologPF, but **not** with `call/N`. The composition of `append` and `map(plus(1))` results in a function which increments the elements of an argument list, and returns a function which prepends that result onto its argument (i.e. `compose(append,map(+1)) @ [[1,2,3]]` $\rightsquigarrow \lambda x \rightarrow$ `append([1,2,3],`x`)`). This abstraction can be passed to `foldr` to be recursively applied to the argument list `[[2],[3,4],[5]]` and `[]` producing `[3,4,5,6]`. The problem that `call/N` has with this example stems from the fact that an intermediate result is produced which is a function abstraction. `Call/N` requires that the right number of arguments must be given for the call to work correctly. For example, `call(plus(1),2,Z)` works correctly giving `Z = 3`, but `call(plus,1,X)` results in an error or fails. This limitation of `call/N` provides the motivation for Naish [58] to recommend `apply/3` in which every application is to one argument and a closure is returned if the function is defined with more.

12. `map(plus,[2,3,4],As)`
    In this case, the application of `map` must return an array of function abstractions, highlighting the weakness of `call/N` as in example 11. PrologPF and `apply/3` both produce the expected result, which can be tested in a query such as
    `?- map(plus,[2,3,4],[Fa,Fb,Fc]), apply(Fb,5,Z).`
    or for PrologPF
    `?- [Fa,Fb,Fc] = map(plus,[2,3,4]), Z = Fb @ [5].`
    giving the solution `Z = 8`.

The examples above illustrate the limitations of `call/N` and show the similarities of `apply/3` and PrologPF for non-deterministic functions. Other examples will highlight the syntactic advantages of PrologPF over `apply/3`, in Table 5.11.

1. `Apply/3` consistently treats all functions as relations, such that the flat form of arithmetic expressions is retained with the `is` relation, as in the example with the definition of `inc`. The functional support in PrologPF allows direct use of nested arithmetic expressions, so the `is` relation is redundant. In fact, if `is` appears in a PrologPF goal with an

	`apply/3`	`PrologPF`
1.	`inc(X,Y) :- Y is X+1.`	`fun inc(X) = X+1.`
2.	`fact(1,1).`	`fun fact(1) = 1;`
	`fact(X,Y) :- X \== 1,`	`fact(N) = N * fact(N-1).`
	`X1 is X-1,`	
	`fact(X1,Y1),`	
	`Y is X * Y1.`	
3.	`apply4(F,A1,A2,R) :-`	`F = plus, Z = F @ [1,2].`
	`apply(F,A1,F1),`	
	`apply(F1,A2,R).`	
	`F = plus, apply4(plus,1,2,Z).`	
4.	`divby2(X,Y) :- Y is X / 2.`	`map(lambda([X],X/2),[2,4,6]).`
	`map(div_by_2,[2,4,6]).`	
5.		`fun div_by_n(N) = lambda([X],X/N).`
		`Z = div_by_n(2) @ [10].`
6.	`fib(0,0).`	`fun fib(0) = 0;`
	`fib(1,1).`	`fib(1) = 1;`
	`fib(N,M) :- N > 1,`	`fib(N) = fib(N-2) + fib(N-1).`
	`N2 is N-2,`	
	`fib(N2,M2),`	
	`N1 is N-1,`	
	`fib(N1,M1),`	
	`M is M2 + M1.`	
7.	`ffib(F,0,M) :- apply(F,0,M).`	`fun ffib(F,0) = F @ [0].`
	`ffib(F,1,M) :- apply(F,1,M).`	`ffib(F,1) = F @ [1].`
	`ffib(F,N,M) :- N > 1,`	`ffib(F,N) =`
	`N2 is N-2,`	`F @ [ffib(F,N-2) + ffib(F,N-1)].`
	`ffib(F,N2,M2),`	
	`N1 is N-1,`	
	`ffib(F,N1,M1),`	
	`MM is M2 + M1,`	
	`apply(F,MM,M).`	

Table 5.6: Further programming examples showing PrologPF capabilities

arithmetic argument, the argument will be evaluated before unification with the corresponding `is` formal parameter. This means that `Z is 1 + 2` $\equiv$ `R = 1 + 2, Z is R`. `is` has quite asymmetric functionality in which the first argument must be a number or a variable while the second argument can also be an arithmetic expression. `(1 + 2) is Z` is not permitted in standard Prolog both for `Z` a variable or with `Z` instantiated to a number. In PrologPF the use of `=` with library functions provides more consistent support for arithmetic, allowing both `Z = 1 + 2` and `(1 + 2) = Z`. The bracketed terms are for clarity, and `1 + 2 = Z` is equally acceptable.

2. The example of the factorial function `fact` shows that deterministic functions in the relational style must have guard conditions in subsequent clauses (i.e. `X` $\backslash ==$ `1`) to prevent erroneous non-deterministic

execution. For more complex functions the conditions can obscure the meaning of the code, and Prolog's *cut* is used to provide an efficient solution. `apply/3` does not attempt to address the presence on `cut` to ensure determinism in functions, while PrologPF has consistent deterministic functional evaluation.

3. The use of `apply/3` provides consistent support for higher-order functional programming, but suffers from the implicit treatment of all function applications as nested applications to one argument and the flat representation of application terms. The example shows the application of an arity/2 function to two arguments, and Naish [58] suggests the definition of an auxiliary relation `apply4` to mitigate this difficulty. PrologPF allows the application of functions to an arbitrary number of arguments in a single term.

4. Without nameless functions, the use of `apply/3` requires that defined functions are created for each requirement, and the chosen name used in the place of the lambda expression in PrologPF. The example shows the specification of a function which divides-by-two. The issue with `apply/3` is mitigated by the use of currying, such that if the required function were times-by-two, then a curried application, for example `times(2)`, could by used. In general, however, an auxiliary fact will be needed, as the example shows.

5. The use of defined functions as an alternative to lambda-expressions with `apply/3` is unsatisfactory where the lambda-expression contains free logical variables. The example shows such an expression in the definition of `div_by_n`, and the issue would similarly arise within a goal such as `?- N = @, Z = lambda([X],X/Z) @ [10].` The implementation with `apply/3` would require the use of the Prolog extra-logical relation `assert` or the accumulation of free variables as additional arguments to the auxiliary functions.

6. The eager argument evaluation semantics of PrologPF is equivalent to the flattened form of Prolog relations used with `apply/3`. The example of the Fibonacci function shows the syntax of PrologPF to be a better match to the requirement.

7. The awkwardness of the flattened form with `apply/3` is exacerbated when nested expressions and higher-order applications appear in the function definition. The example gives a modified Fibonacci function where an additional parameter specifies a function (`F`) to be applied to the sub-terms before summation in the recursive case.

The `apply/3` example of `ffib` illustrates the following differences with PrologPF:

(a) The condition `N>1` is required as `apply/3` has no special consideration for deterministic execution, and assumes use of additional conditions or *cut*.

(b) All expressions with `apply/3` retain their flat Prolog form, leading to an unwieldy syntax for expressions which would naturally be nested.

(c) Arithmetic with `apply/3` relies upon the use of the special Prolog relation `is`. In PrologPF arithmetic expressions can appear anywhere as a valid argument term, and will be reduced before the term is unified with the corresponding formal argument.

(d) Function application in PrologPF can be either explicit with the `@` operator, or implicit by using a defined function name in a compound argument term. The latter case is defined to be syntactic sugaring for the former. The definition of `ffib` using `apply/3` differentiates between the application of a higher-order term represented by the variable `F` in `apply(F,MM,M)` and the recursive call to the function `ffib` in `ffib(F,N2,M2)`. For consistent use of `apply/3`, the recursive call would be replaced by `apply(ffib(F,N2),M2)` which would be converted by `apply/3` to the call `ffib(F,N2,M2)`. It is unclear whether it is better to make consistent use of `apply/3` in higher-order functions and render the non-curried calls more obscure, or whether a mix of `apply/3` and normal relation calls should be used.

5.12 Conclusions

Higher-order functions can be neatly integrated with Prolog's relations with a deterministic evaluation semantics compatible with the requirements of a Delphi implementation.

Examples given in this and the following two chapters show the capabilities chosen for implementation in PrologPF to be sufficient to express a wide range of programs without resorting to artificial or obscure coding devices.

The capabilities of PrologPF, including the definition and application of functions, the call-once semantics of the `if` condition, and the use of boolean functions as relations, have proved sufficient to preclude the need for *cut* in all the test programs reviewed to date.

5.13 Summary

The functional component of PrologPF extends the Prolog language in the following ways:

- The definition of functions through the `fun` relation
- The application of functions through the `@` operator
- Support for higher-order functional programming through the use of lambda-expressions and currying
- A general strict functional evaluation semantics with the single exception of a pre-defined `if` function
- Use of relations within functions is limited to the condition argument of the `if` function, where deterministic search for the first solution is enforced
- Support for boolean functions to be treated as relations

These features have proved consistent in use and sufficient to implement a wide range of sample programs without resorting to *cut*.

The defined semantics permit an efficient implementation on an extended Delphi Machine, where function applications embedded within a Prolog program are compiled to direct machine-code calls. Such an implementation has been produced in PrologPF.

Chapter 6

Case Studies: Foundations of Computer Science Exercises 4 and 6, and the Prolog Technology Theorem Prover

This chapter provides an analysis of the use of PrologPF in expressing algorithms typical of higher-order functional programming, and testing the use of PrologPF on a substantial problem. The sample programs have been taken from the functional programming exercises in Standard ML [55] set to undergraduate students on a university Computer Science class [60], and the Prolog Technology Theorem Prover from SRI [70].

The first example, Exercise 4, introduces the style of functional programming in PrologPF and shows the use of a Prolog relation within a function definition. The second example, Exercise 6, uses higher-order functional programming to create infinite lists, and shows how this style of functionally implemented data structure can be used within a relational program. The Prolog Technology Theorem Prover, with a large test problem, provides a suitable body of Prolog code for transferral to PrologPF, for potential parallel speedup.

6.1 Foundations of Computer Science Exercise 4 - Routes

6.1.1 Problem Description

The sample problem is to form the transitive closure of a relation. A function `routes` is to be defined which, given a list representing the arcs of an acyclic graph, returns a similar list representing all possible connections. The initial list representing the graph can be [(a,b),(b,c),(b,d),(d,e)], for the graph given in Figure 6.1.

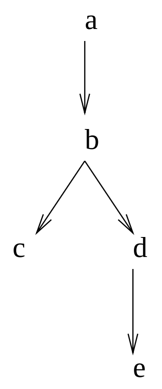


Figure 6.1: Initial acyclic directed graph for Exercise 4.

The function call `routes([(a,b),(b,c),(b,d),(d,e)])` should return the expanded list `[(a,c),(a,d),(a,e),(b,e),(a,b),(b,c),(b,d),(d,e)]`, representing the graph in Figure 6.2.

6.1.2 Startpoints and Endpoints

The final algorithm will use some utility functions to produce intermediate results. Firstly, we define the function `startpoints` which, given a list of arcs and a node, will return the sublist of arcs with that node as the endpoint.

```
fun startpoints([],Z)             = [];
    startpoints([(X,Z)|Pairs],Z) = [X|startpoints(Pairs,Z)];
    startpoints([(X,Y)|Pairs],Z) = startpoints(Pairs,Z).
```

The functional syntax of PrologPF is clearly similar to Standard ML [55]. The eager evaluation semantics are also similar, but PrologPF is typeless.

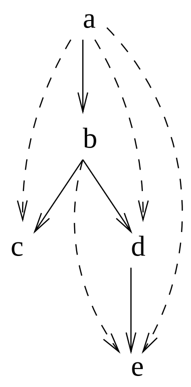


Figure 6.2: Complete graph for Exercise 4.

The Prolog syntax for variables and lists has been preserved. The function definition is terminated with a full stop (.), allowing the definition to be a readable term in standard Prolog [35]. The functional support in PrologPF can be added to a Prolog compiler without change to the parser. A PrologPF program without functions has the same syntax as an identical Prolog program. The PrologPF definition of `startpoints` has exploited the use of a logical variable `Z` in the second case. The definition assumes an ordering of the cases, with the first left-hand-side to successfully unify with the arguments in the call being deterministically selected for the next reduction step. The function could be written with an *if-then-else* expression replacing the second and third cases, avoiding the use of logical variables, in which case the code would be almost identical to the same function written in ML.

To produce a sublist of endpoints from a given startpoint, a similar function `endpoints` is needed:

```
fun endpoints([],X)             = [];
    endpoints([(X,Y)|Pairs],X) = [Y|endpoints(Pairs,X)];
    endpoints([(Z,Y)|Pairs],X) = endpoints(Pairs,X).
```

6.1.3 Allpairs and append

Here we develop the function `allpairs` which produces the Cartesian product of two lists of pairs. The function will use a utility function `append`, which returns the list formed from appending its two actual arguments:

```
fun append(X,Y) = if append(X,Y,Z) then Z.
```

The PrologPF *if-then-else* expression has a relational goal as the condition. If the goal succeeds (with an associated unifier) then the result is the *then* expression, otherwise it is the *else* expression. The *if-then* form used in the example has an implicit *else* expression of `fail`. The example of `append` shows the straightforward mapping of a particular mode of a relation into an equivalent function. PrologPF distinguishes between functions and relations of the same name if they have differing numbers of arguments. Curried use of the PrologPF function `append` is permitted, where the function call will have fewer than two actual arguments. In the simplest definitions, such as `append`, the equivalent relation will often have one *more* argument than the equivalent function. This enables PrologPF to provide a set of library functions representing many of the relations expected in a standard Prolog library, using the same names.

The first stage to provide the Cartesian product of two lists is to define a function which pairs one element with every value of a list, producing a list of pairs. This function, `pairx` is defined as follows:

```
fun pairx(_,[])      = [];
    pairx(X,[Y|Ys]) = [(X,Y)|pairx(X,Ys)].
```

The function `pairx` illustrates the use of "_" to represent an anonymous variable, consistent with the syntax in both Prolog and ML. The function `allpairs` also uses an anonymous variable, and the functions `append` and `pairx`:

```
fun allpairs([],_)       = [];
    allpairs([X|Xs],Ys) = append(pairx(X,Ys), allpairs(Xs,Ys)).
```

In common with the global definition of relational procedures, PrologPF has no support for the lexical scoping of function definitions. The equivalent functions in ML could be nested to place the value of `x` in `allpairs` within the scope of `pairx`:

```
fun allpairs([],_)        = [];
    allpairs((x::xs),pairs) =
        let
           fun pairx([])     = [];
               pairx(y::ys) = (x,y)::pairx(ys).
        in
           pairx(pairs) @ allpairs(xs,pairs)
        end;
```

The ML definition also takes advantage of the infix definition of the ML library append function `@`.

6.1.4 Addnew

We can call a list of arcs *complete* if whenever it contains two arcs `(a,b)` and `(b,c)` then it also contains the arc `(a,c)`. The function `addnew`, given an arc and a complete list of arcs, will return a complete list including the new arc. For example, `addnew((a,b),[(b,c)])` will return `[(a,b),(b,c),(a,c)]`. Functions `addall` and `addnew` are mutually recursive. Firstly, the function `addall` uses `addnew` to insert each arc in its first argument into the complete list given as its second.

```
fun addall([],Pairs)       = Pairs;
    addall([P|Ps], Pairs) = addall(Ps, addnew(P,Pairs)).
```

The function `addnew` has an arc as its first argument and a complete list of arcs as its second.

```
fun addnew(Pair,[])      = [Pair];
    addnew((X,Y), Pairs) =
        if (member((X,Y),Pairs); X=Y)
        then Pairs
        else addall( append( allpairs( startpoints(Pairs,X), [Y]),
                             allpairs( [X], endpoints(Y,Pairs))),
                    [(X,Y)|Pairs]).
```

In the general case, `addnew` will prepend the new arc `(X,Y)` onto the complete list of arcs given as a second argument, and will call `addall` to recursively add all the arcs leading to `X` or leading from `Y`. The example *if-then-else* expression in `addnew` further illustrates the use of a relational goal as the condition, in which the disjunctive operator ; is used to represent `orelse`. The operator has the same left-to-right interpretation as in sequential Prolog.

6.1.5 Routes

Finally the function `routes`, given an arbitrary list of arcs as its argument, will recursively call `addnew` to add each arc to an accumulated complete list:

```
fun routes([])            = [];
    routes([(X,Y)|Pairs]) = addnew((X,Y),routes(Pairs)).
```

The function `routes` can be exercised by its use in a top level goal such as:

```
           :- Z = routes([(a,b),(b,c),(b,d),(d,e)])
```

The goal will succeed with the single solution,

```
    Z = [(a,c),(a,d),(a,e),(b,e),(a,b),(b,c),(b,d),(d,e)]
```

6.2 Foundations of Computer Science Exercise 6 - Primes

This exercise exploits the higher-order programming capabilities of PrologPF to simulate lazy execution for the definition of infinite lists. A function `primes` is defined using a sieve algorithm to return an infinite list of primes. This function is used in the simple definition of a relation `prime(P)` which succeeds for prime `P`, and can be used as a generator for primes within a relational goal.

6.2.1 Infinite lists

A carefully designed PrologPF term can be used to represent an infinite list. The term can be a compound term `item(X,Xf)` where `X` is the value to be found at the head of the list, and `Xf` is a *function* which can be called to return the tail of the list. The tail of the infinite list will itself be a compound term of the same structure.

Thus, for example, the infinite list of the natural numbers can be represented by the PrologPF term:

```
    item(1, lambda([],item(2,lambda([],item(3,...)))))
```

This term can be constructed by the function `makeints`:

```
fun makeints(N) = item(N, lambda([], makeints(N+1))).
```

6.2.2 Head, tail and nth

As with the usual Prolog-style lists, functions such as `head` and `tail` can be created which extract the components of the infinite lists:

```
fun head(item(I,_)) = I.

fun tail(item(_,Xf)) = Xf @ [].
```

The `tail` function uses the explicit application operator `@` to evaluate the function representing the tail of the list. With the example of the term representing the infinite list of integers given above, `tail` will evaluate:

```
              lambda([],item(2,lambda(..)))  @ [],
```

returning the term: `item(2,lambda(..))`.

An additional utility function common in the use of lists is `nth`, with `nth(S,N)` returning the $\mathtt{N}^{th}$ element of a list `S`:

```
fun nth(S,1) = head(S);
    nth(S,N) = nth(tail(S),N-1).
```

The definition of `nth` illustrates the use of `head` and `tail` to abstract the definition of the term used to represent the infinite list. The function `nth` can be used in a curried form, where `nth(S)` represents a function which when applied to an integer will return the element of the list `S` indexed by that integer.

6.2.3 Filters

The sieve algorithm used to produce the infinite list of primes requires a higher-order function `filters` which, given a selection function and an infinite list, returns the list containing those elements for which the selection function is true.

An example of a selection function, and that used in `primes`, is `notdiv`, returning `true` if the first argument is not an exact divisor of the second:

```
fun notdiv(X,Y) = if (Y mod X =\= 0) then true else false.
```

The function uses the Prolog library relation `=\=` and the Prolog arithmetic function `mod`, as an alternative to defining these as PrologPF functions for a simpler boolean definition of `notdiv`.

The function `filters` applies the selection function given as the first argument in the condition of an *if-then-else* expression. The condition is interpreted as a relational goal, and `filters` exploits the PrologPF treatment of the function call `F @ [X]` in this position as the goal `(F @ [X]) = true`.

```
fun filters(F, item(X,Xf)) =
        if (F @ [X])
        then item(X,lambda([], filters(F, Xf @ [])))
        else filters(F, Xf @ []).
```

The function illustrates the use of higher-order variables to represent functions, the explicit application of those functions using the operator `@`, and the creation of nameless functions using `lambda`.

6.2.4 Primes

Given the definition of the term used to represent an infinite list, and the utility functions defined above, the definition of the function `primes` is straightforward:

```
fun primes(item(X,Xf)) =
      item(X, lambda([], primes(filters(notdiv(X), Xf @ [])))).
```

Given an infinite list of integers starting with a prime, the function will return the infinite list beginning with that number, followed by the infinite list of the application of `primes` to the list having filtered out all the elements divisible by the first prime. The encapsulation of the tail of the list within the `lambda` expression serves to delay the evaluation of the tail, avoiding an infinite loop.

The functional definition and creation of infinite lists in PrologPF has a natural use within relations. Firstly, a relation `next_prime` can be defined which succeeds for each element in a list:

```
next_prime(Primes,P) :- P = head(Primes).
next_prime(Primes,P) :- next_prime(tail(Primes),P).
```

Finally, the relation `prime(P)` can be defined which succeeds for each prime integer `P`, and provides a generator for the primes:

```
prime(P) :- next_prime(primes(makeints(2)), P).
```

A top level goal `:- prime(P)` provides an infinite sequence of solutions `P=2, P=3, P=5,...`.

6.3 Prolog Technology Theorem Prover

The Prolog Technology Theorem Prover (PTTP) developed by Mark Stickel [70] improves upon the incomplete semantics of Prolog by:

1. Using a sound unification algorithm with an occurs check.
2. Permitting general logical formulas to be used in clauses, rather than just the Horn clauses accepted by Prolog.
3. Replacing the unsound depth-first search with an iterative deepening search.

PTTP also retains information on which formulas where used for each inference so that the proof can be printed. The theorem prover transforms the assigned problem into a suitable Prolog program, which is then compiled and executed using a standard Prolog compiler. PTTP is approximately 1500 lines of Prolog code, including comments, divided into the following parts:

1. The code to transform the general assigned problem into a suitable Prolog program.
2. The utility relations used by that program to ensure its sound execution, for example the unification algorithm and various list utility predicates.
3. The clauses representing the sample problems.

The sample problems used in the case study result in Prolog programs of 400 to 500 lines of Prolog code, including the needed utilities.

6.3.1 Chang and Lee example 2

As an example of the execution of the Prolog Technology Theorem Prover, the first sample problem is taken from [21]:

```
p(e,X,X).
p(X,e,X).
p(X,X,e).
p(a,b,c).
p(U,Z,W) :- p(X,Y,U), p(Y,Z,V), p(X,V,W).
p(X,V,W) :- p(X,Y,U), p(Y,Z,V), p(U,Z,W).
query :- p(k(X),X,k(X)).
```

A Prolog term representing this problem is transformed into a Prolog program which is run producing the proof in Figure 6.3.

The Prolog program representing the transformed problem contains a small number of relations using *cut*, which can be replaced with PrologPF functions. For example, the relation `identical_member` can be replaced with a function:

```
identical_member(X,[Y|_])  :-
      X == Y,
      !.                        % note presence of cut
identical_member(X,[_|L]) :-
      identical_member(X,L).
```

```
Goal#  Wff#  Wff Instance
-----  ----  ------------
  [0]    7   query :- [1].
  [1]    5      p(b,a,c) :- [2] , [9] , [10].
  [2]    6         p(c,a,b) :- [3] , [4] , [8].
  [3]    3            p(c,c,e).
  [4]    5            p(c,b,a) :- [5] , [6] , [7].
  [5]    4               p(a,b,c).
  [6]    3               p(b,b,e).
  [7]    2               p(a,e,a).
  [8]    1            p(e,b,b).
  [9]    3         p(a,a,e).
 [10]    2         p(c,e,c).
```

Figure 6.3: Solution for Chang and Lee example 2.

The equivalent function in PrologPF which avoids the use of *cut* is as follows:

```
fun identical_member(X,[]) = false;
    identical_member(X,[Y|L]) =
        if (X == Y)
        then true
        else identical_member(X,L).
```

The compiled PrologPF version of the Chang Lee example completes on a single cpu with the PrologPF partitioning depth limit set to 1 (i.e. a single partition) in 2187 milliseconds. The execution on the distributed PrologPF system with a suitable depth limit $L = 23$ for parallel execution produces the runtimes of the graph in Figure 6.4.

The problem is sufficiently small to limit the runtime to a minimum of about 500 milliseconds. A graphical representation of the search tree is given in Appendix B.3.

6.3.2 Overbeek example 4

Another example, provided by Overbeek [37], provides a much greater search space and potential for much improved parallel speedup when compiled with PrologPF. The example problem provided by Overbeek has a single cpu runtime on the processors used by the distributed PrologPF system of over five hours:

```
p(e(X,e(e(Y,e(Z,X)),e(Z,Y)))).
p(Y) :- p(e(X,Y)), p(X).
query :- p(e(e(e(a,e(b,c)),c),e(b,a))).
```

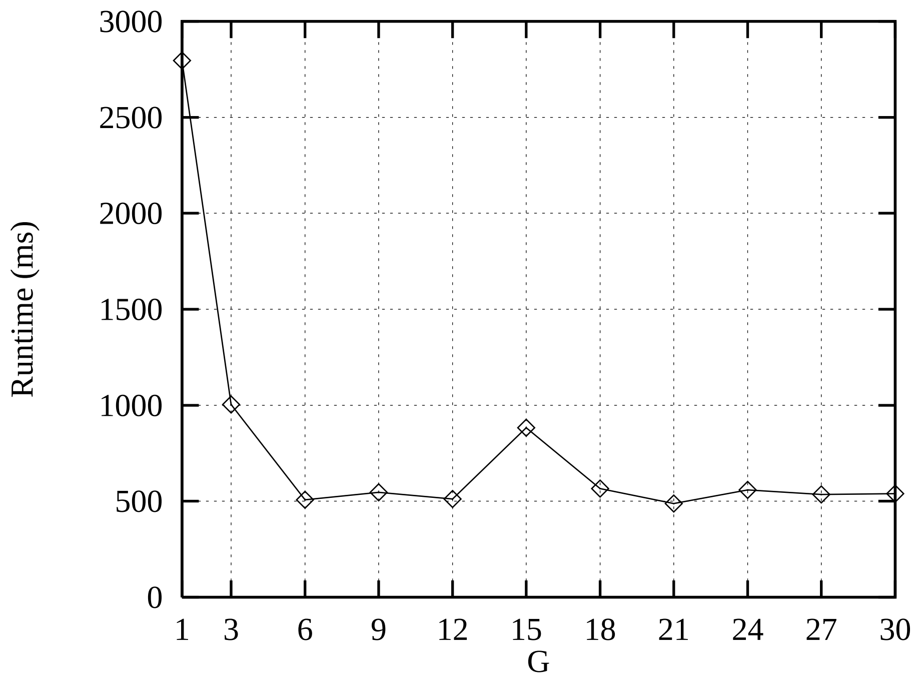


Figure 6.4: Runtimes for Chang Lee example 2 for $G = 1 \ldots 30$ and $L = 23$.

Compilation and execution proceeds in the same manner as for the Chang and Lee example. The graph given in Figure 6.5 shows the improvement in runtime as processors are added to the group used to execute the problem.

The improvement is clarified with the speedup graph in Figure 6.6 which plots the speedup ratio against the single cpu case for groups of path processors up to a maximum group size of 42.

The speedup graph shows linear speedup throughout the range of group sizes available. For some values of G, such as 18 and 27, the speedup is greater than the increase in the number of path processors. This phenomenon is a result of the single-solution requirements of the code produced by PTTP. In a sequential execution, the first solution found will be that furthest to the left in the depth-first, left-to-right search tree. Also, the search tree to the left of that solution will be fully search before the solution is discovered. In the distributed execution of PrologPF, the search tree is partitioned between the available path processors and the first solution found will be that furthest to the left *within the subtree assigned to that path processor.* The partitioning may result in a solution appearing very early in the subtree assigned to one of the path processors, such that it is found very quickly. The situation is

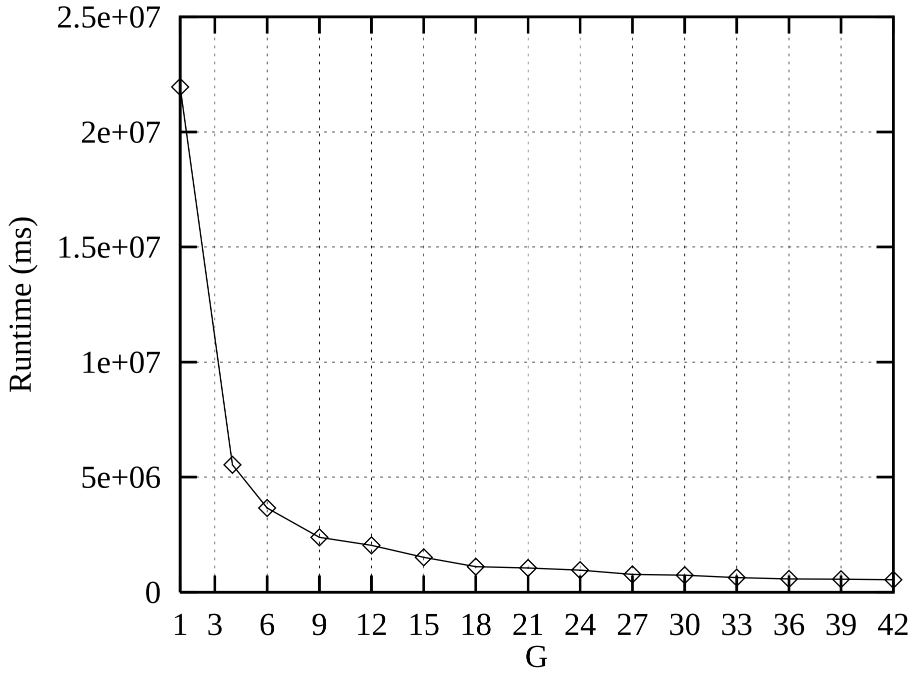


Figure 6.5: Runtimes for Overbeek example 4 for $G = 1 \ldots 42$ and $L = 130$.

illustrated in the simplified diagram in Figure 6.7.

In the single-cpu case, the subtrees labelled A and B in the figure will be searched first, taking at least 520ms, and then the solution `X` will be discovered after at least 2ms in the subtree labelled C. If the problem is divided between three path processors, then oracles leading to A and D will be allocated to path processor 0, oracles B and D to processor 1, and oracle C to processor 2. Path processor 2 will discover the solution `X` after the partitioning time (20ms) followed by the search to the solution in subtree C, taking approximately 52ms. In this simple example, the shallow positioning of the solution in subtree C gives a speedup for the single-solution case of approximately 520/52, i.e. 10, with only 3 path processors. Note that for the all-solutions case, the subtrees A,B,C,D and E, would all have to be fully searched such that the benefit of one or more shallow solutions will not be obtained. The one-solution requirement of the Prolog Technology Theorem Prover means that with fortuitous partitioning greater than linear speedup can be obtained.

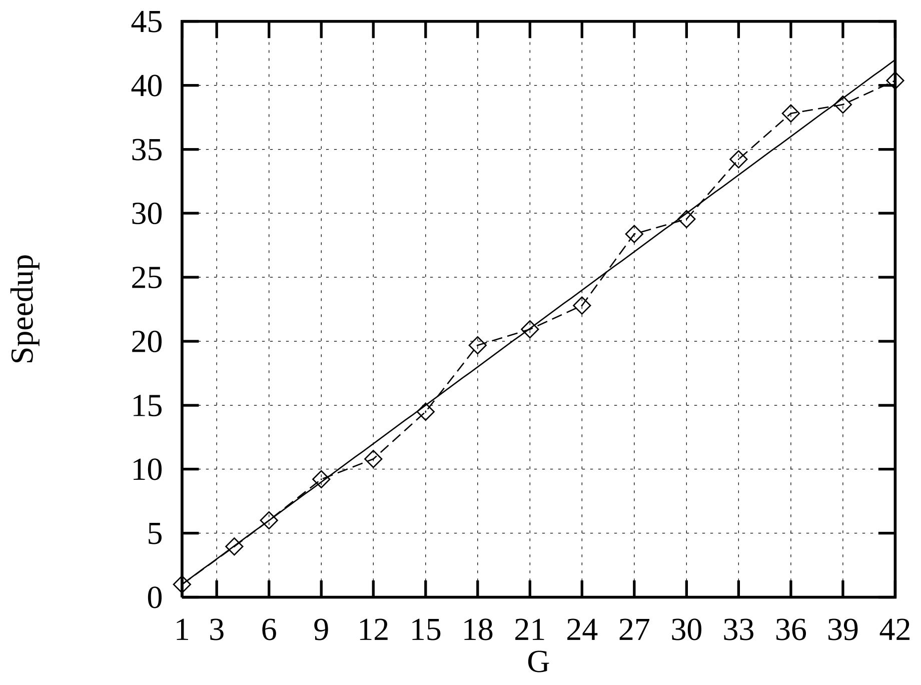


Figure 6.6: Speedup for Overbeek example 4 for $G = 1 \ldots 42$ and $L = 130$.

6.4 Conclusions

PrologPF provides support for functional programming comparable to that of a typeless ML, in which some programs have a more convenient expression than their Prolog equivalents. The deterministic reduction of functional expressions permits the expression of many algorithms that would otherwise suggest the use of *cut* in Prolog.

The syntax for functional definition and evaluation in PrologPF is consistent with the standard Prolog syntax for the relational procedures, such that functions and relations can be mixed in a program without conflict of programming styles. Data structures, such as lists and compound terms, are common to the relational and functional components of PrologPF.

The combined functional and logic support in PrologPF, with the relational procedures compatible with standard Prolog [35], allow a straightforward conversion of substantial Prolog programs to equivalent PrologPF programs suitable for execution on the Delphi Machine. In the case of the largest program tested, the Prolog Technology Theorem Prover, a speedup of 40 times using 42 processors was achieved.

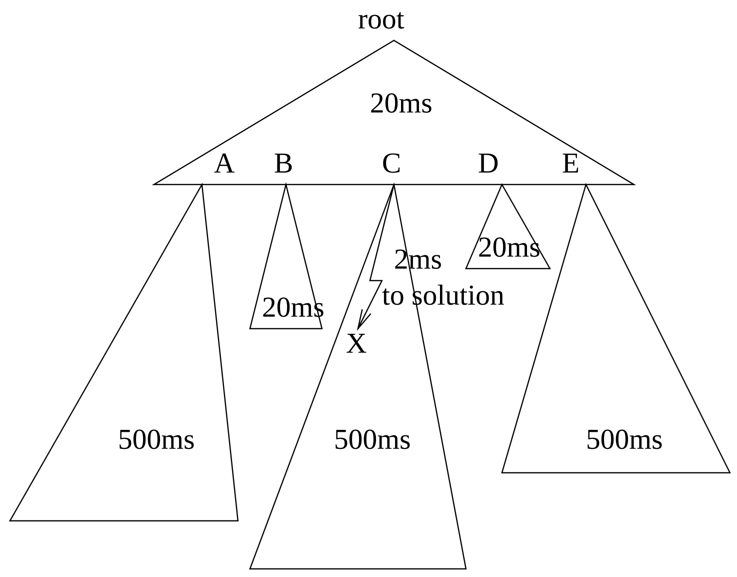


Figure 6.7: Greater than linear speedup for single-solution problems.

6.5 Summary

The problem of finding the transitive closure of a relation provides a suitable exercise for functional programming, with the relation defined as a set of arcs between nodes of a graph. Prolog terms can be used to represent the arcs (i.e. the tuple `(a,b)` to represent the arc $a \rightarrow b$), and the set of arcs stored in a Prolog list. The functions necessary to transform this list into a complete list representing the transitive closure can be implemented in PrologPF in as straightforward a manner as in Standard ML [55]. Functions and terms are typeless in PrologPF, consistent with the typeless environment of Prolog. Potential benefits of a polymorphic type inferencing system as found in ML have not been provided in PrologPF.

The higher-order programming capabilities of PrologPF include the definition of nameless functions using `lambda` expressions, and the ability to pass functions as arguments and return them as results. These capabilities can be demonstrated in the use of lazy evaluation in the implementation of infinite lists. The terms used to represent the lists contain lambda expressions, and higher-order functions `head` and `tail` are used to extract the elements of the list. A function `primes` can be written to return the infinite list of primes. The example given shows the integration of the functional support with a relation `prime(P)` which succeeds for any prime, P. The relation can be used with `P` an integer, or a `P` a variable which will be instantiated with a sequence of primes.

The Prolog Technology Theorem Prover is a Prolog program which can transform the predicate calculus representation of a logic program into an equivalent Prolog program avoiding some incomplete aspects of Prolog's execution. Depth-first search is replaced with breadth-first, and an explicit occurs check is embedded in the code. The procedures in the program containing *cut* are replaced with equivalent functions, and the resulting program compiled with the PrologPF compiler for a speedup in execution of up to 40 times on the Cambridge laboratory's 42 workstations.

Chapter 7

Kappa: a simplified parallel logic processing primitive

This chapter gives an analysis of the implementation of the breadth-first partitioning scheduling strategy without using oracles. The oracle-based strategy first proposed by Clocksin and Alshawi in [28] and used in PrologPF for comparison with DelphiKS[1] in Chapter 3 is compared with a simplified technique using a special proposition `kappa`.

The novel partitioning proposition `kappa` is suitable for use with any standard Prolog compiler. While the technique has much in common with the breadth-first partitioning strategy of PrologPF, `kappa` can be implemented without the use of oracles. The limitations of the new primitive are compared with the strategy improvements potentially available through the improved exploitation of oracles.

7.1 Background

Chapter 3 reviews in detail the breadth-first partitioning strategy using oracles to define the root of each subtree to be allocated to an available path processor.

In the first phase of execution, the search is bounded by a selected depth limit L. The open branches found at this limit are recorded as a count S of *oracles*, each representing the sequence of clauses used to arrive at the point in the search at which the depth limit was reached. An *oracle stack* is used to accumulate the open oracles during this first oracle discovery phase. While

[1]the previous implementation of the Delphi Machine, documented in [49, 66].

the search in the initial phase is bounded by L, the standard Prolog depth-first left-to-right search is used, and the S open oracles form an ordered list in the order of discovery. Figure 7.1 shows an example subtree traversed during the depth-limited initial phase, with the resultant oracle stack as a data structure representing the paths in the reduced tree.

The oracles in the oracle stack can be allocated to a number of *path processors* whose role is to follow each assigned oracle to recreate the environment required at the root of the associated subtree at depth L, and then to continue the search of that subtree.

The *breadth-first partitioning strategy* used by PrologPF proceeds in the two phases of oracle discovery and subsequent subtree search. While the model supports the use of a single control processor for the execution of the first phase and then the allocation and communication of the oracles, a distributed model is used in PrologPF in which all the path processors execute the first phase and create a local copy of the oracle stack. The path processors then use the parameters G and N representing the processor group size and the unique processor number respectively to select disjoint subsets of oracles from the oracle stack.

7.2 Breadth-first partitioning without oracles

If the two phases of the breadth-first partitioning algorithm are interleaved, it is possible to create a similar one-time partitioning strategy without the use of oracles. PrologPF completes the first oracle discovery phase before assigning the open oracles to the available path processors. In practice the oracle assignment function is fixed before the start of execution, such that the assignment can be performed independently on each path processor. However, with reference to Figure 7.1, on discovery of the first open oracle at the depth limit L the Prolog stacks and heap contain the environment necessary for the continued search of the subtree labelled A in the figure. This is the environment recreated when the oracle is followed during the second phase of BFP. With a suitable filtering function applied at the depth limit L, the subtrees can be selected and searched by the path processors *as the open branches at L are discovered.* The principles of this strategy without oracles are as follows:

- The phases equivalent to the initial oracle discovery and subsequent subtree search in PrologPF are interleaved.
- Execution proceeds without maintenance of the current oracle, instead exploiting the environment constructed during the search beneath the

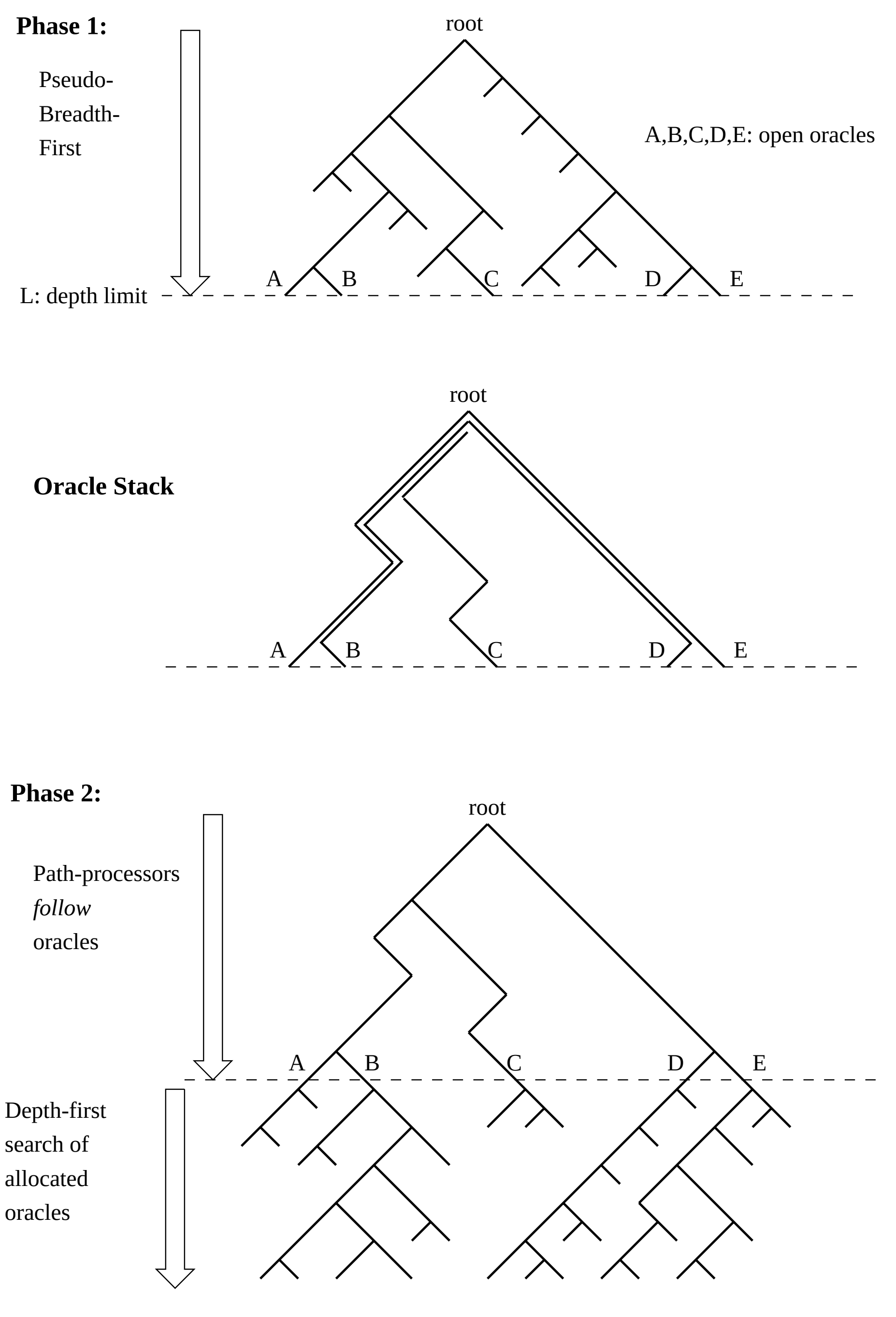


Figure 7.1: Use of oracles in breadth-first partitioning.

depth limit.

- Each path processor executes the search from the root of the search tree, with the following assigned parameters:

  G: The number of path processors in the group.

  N: The unique processor number of the given path processor.

  L: The selected depth limit.

- The search is generally limited to the depth set by L, with a count accumulating the number of times this depth limit is reached during the depth-first left-to-right search. This count is equivalent to the oracle number in the ordered stack resulting from the first phase of BFP. The point in the search tree at which an open branch is discovered at the depth limit L is called a *port*. Each time the count is incremented, the selection function is applied to determine whether the search should continue with the subtree beneath the port, or whether the port should be skipped.

- A suitable selection function will ensure that all ports are selected by the combined group of path processors, and no port is selected more than once. As with the BFP algorithm in PrologPF, a good selection function would allocate the work beneath the ports evenly. In the absence of a work estimation function, an equal number of ports can be allocated to each path processor.

Figure 7.2 shows a search tree during the execution of a scheduling strategy similar to BFP without using oracles. The ports are identified at the depth limit L, and the selection function at a given path processor is illustrated.

The path processor executes the search bounded by the depth limit until a port is accepted by the selection function, at which point the search continues with the subtree beneath that port. On completion of that subtree, the search continues bounded by the depth limit and the selection function.

7.3 The gate proposition `k_gate`

The selection function described in Section 7.2 can be represented by a proposition tested each time a subgoal is called or executed. If the current depth is not that of the depth limit, then the proposition succeeds. If the depth limit has been reached, then the proposition fails unless the current port is assigned to the given path processor.

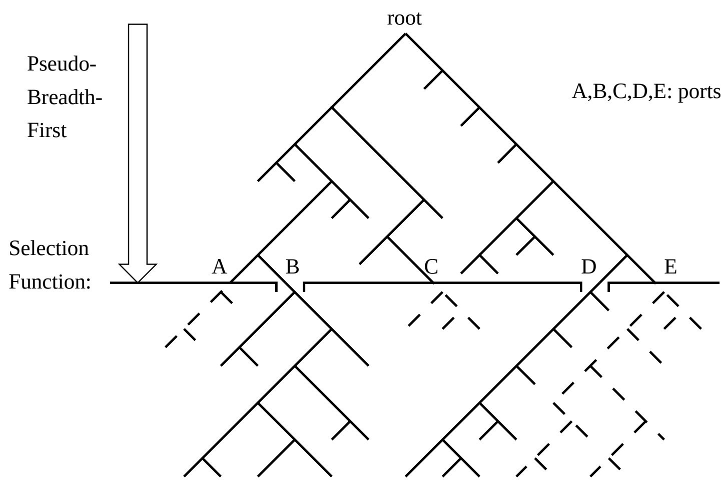


Figure 7.2: Search tree partitioning without oracles.

One suitable function which will evenly allocate the ports to the available path processors is:

$$(\text{port\_number mod } G) = N$$

The selection of the ports *modulo* G means that the path processors search the subtree beneath every Gth port, beginning with the Nth. Thus with G path processors, allocated values of N from 0 to $G-1$, all ports will be uniquely allocated.

A pseudo-Prolog proposition providing the port selection behaviour using this function is `k_gate`, given below:

```
k_gate :- current_depth <> L.
k_gate :- current_depth = L,
          increment port_number,
          (port_number mod G) = N.
```

The proposition accesses global values for the current depth, the assigned depth limit L, and `port_number`, the number of ports found so far. `k_gate` has the side-effect of incrementing `port_number` each time the depth limit is reached. `port_number` is initially 0.

7.4 kappa

kappa is the proposition representing the parallel partitioning primitive inserted into the user program. In the absence of an existing global system value representing the search depth, kappa can accumulate a depth value and call k_gate to determine success or failure. depth is initially 0.

```
kappa :- increment depth, k_gate.
kappa :- decrement depth, fail.
```

In the example of the member/factorial program given in Section 3.7 the user code can be modified to use kappa as follows:

```
member(X,[X|_]).
member(X,[_|Y]) :- kappa, member(X,Y).

fact(1,1).
fact(N,F) :- kappa, N > 1,
             kappa, N1 is N - 1,
             kappa, fact(N1,F1),
             kappa, F is N * F1.

:- kappa, member(X,[4,3,2,1]), kappa, fact(X, F).
```

Preceding every goal with a call to the proposition kappa can be achieved automatically through the use of the utility relation term_expansion/2 provided with many Prolog implementations [16]. If kappa is only used at selected positions in the program, then similar behaviour is achieved with a new meaning for the *depth* value used in the selection algorithm. In this case, the depth value is taken to mean the depth of the calls to kappa, which is no longer equal to the current depth in the AND-OR search tree.

Many implementations of Prolog provide a mechanism to incorporate C code into procedure definitions. This capability permits efficient implementation of the kappa primitive, integrating the function of k_gate, resulting in the final definition of kappa given in Table 7.1.

7.5 Kappa at every subgoal versus selective use

If the special proposition kappa is inserted before every subgoal in the user program, the global depth value updated by the frequent calls to kappa is equal to the current depth of the search into the AND-OR tree of the

```
kappa  :-  if ((++depth == L) && ((++port_number mod G) == N))
           then succeed
           else fail.
kappa  :-  --depth, fail.
```

Table 7.1: Definition of parallelisation primitive `kappa`.

original program. With the example of the `member/factorial` example program given in Chapter 3, Figure 3.21, the selection function provided by `kappa` can be viewed as a horizontal boundary with ports at a constant AND-OR depth. The situation is illustrated in Figure 7.3

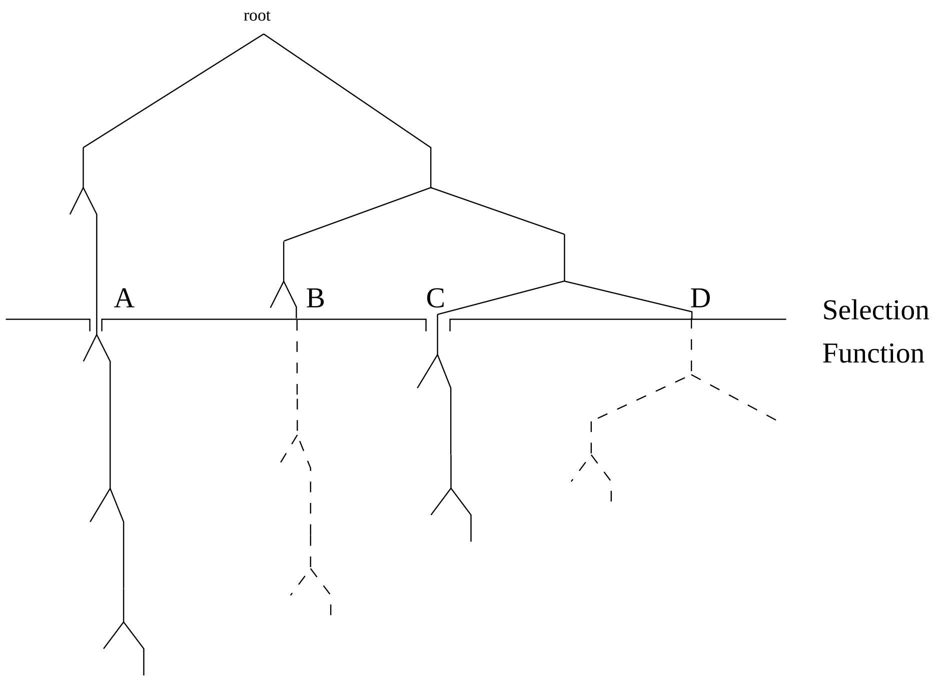


Figure 7.3: Search tree for `member/fact` program with horizontal selection function.

However, `kappa` can be more selectively added to a user program for similar benefits without the overhead of a call to the proposition before every original subgoal.

An example of the selective use of kappa in the member/factorial program is:

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).

fact(1,1).
fact(N,F) :- N > 1,
             N1 is N - 1,
             kappa, fact(N1,F1),
             F is N * F1.

:- member(X,[4,3,2,1]), kappa, fact(X, F).
```

In the example above, only the fact relation is associated with the kappa proposition. The depth value maintained by kappa is now purely the depth into the fact relation, and a diagram representing the search tree is given in Figure 7.4.

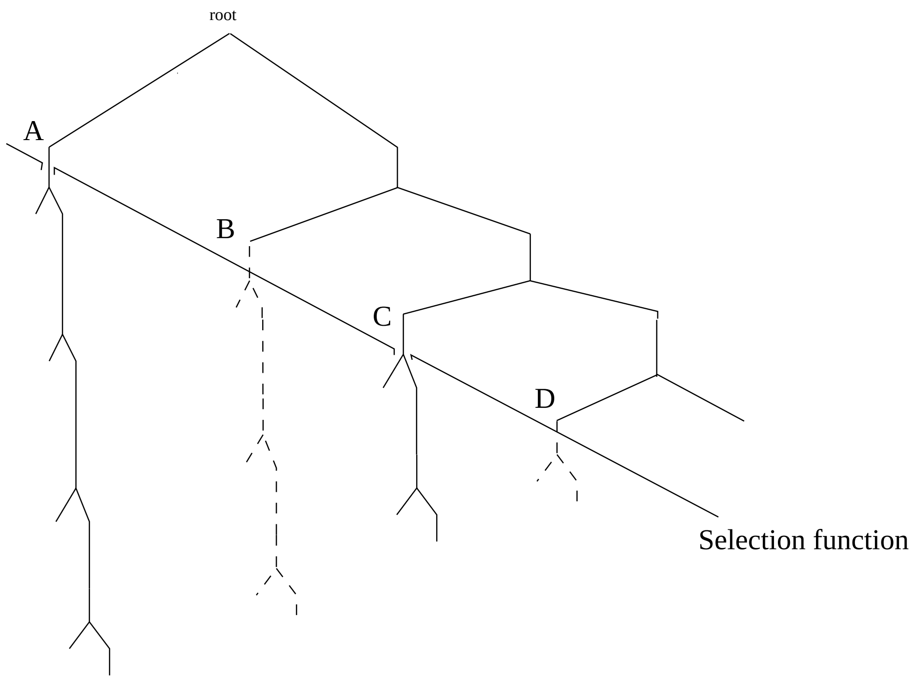


Figure 7.4: Search tree for member/fact program with sloping selection function.

With selective use of kappa, the programmer (or compiler) would select those relations known to generate a large number of branches near the root of the search tree. In the minimal case, kappa might be associated with just one relation in the user program, and an appropriately small value of the depth limit would be used. The `member/fact` program given above will partition the work satisfactorily with `kappa` associated with the relation `fact` and a depth limit set at 1. Higher values of the depth limit L will not improve the partitioning of the problem as the `fact` relation is deterministic and partitioning deeper into the `fact` relation will result in the same number of ports.

A more general example of the selective use of `kappa` is given in the procedure `findsum` below [2].

```
% L is given list of integers
% S is given sum
% [X|Y] is subset of L summing to S

findsum(L,S,[X|Y]) :- select(L,X,L1),
                      S1 is S - X,
                      kappa, findsum(L1,S1,Y).
findsum(_,0,[]).

:- findsum([1,2,3,5,7,11],14, X).
```

If the depth limit L for the execution of the problem is set at 1, then partitioning will take place after selection of the first candidate integer for the sum. For $L = 2$, partitioning will take place after the selection of two integers, and so on.

7.6 Efficiency considerations

7.6.1 Sequential computation when depth $< L$

PrologPF performs the initial pseudo-breadth-first search without advantage from the parallel processing available in the distributed system. Two techniques are available to PrologPF:

1. Perform the initial search in the control processor, and communicate the discovered oracles to the available path processors.
2. Duplicate the initial search in every path processor, such that an oracle stack is held locally on every machine. The oracles can be allocated using a common algorithm ensuring each oracle is allocated to one path processor, and all oracles are allocated.

[2] `select(L,X,L1)` is a library relation with X a value from list L with the remainder of the list in L1.

The implementation used as the subject of this dissertation uses the second technique, but has been tested with the first. The earlier implementation using oracles, DelphiKS [49, 66], also used both techniques.

The parallel speedup available with the use of `kappa` is similarly limited by the sequential search of the tree beneath the depth limit L. However, the use of `kappa` requires that the second technique listed above be used. The interleaving of the depth-limited search with the subsequent subtree search and the lack of oracles means that each path processor must proceed with the pseudo-depth-first search to create the environment required at each port before searching an assigned subtree.

The duplicated processing (or equally the sequential processing of the first technique) imposes a limit on the maximum speedup of the problem. For PrologPF this issue is discussed in Chapter 3, Section 3.6.1. If the depth limit is set too large, then a large proportion of the total search tree may reside within the depth limit L, while only the subtrees at depths greater than L are available for parallel search.

7.6.2 Optimal selection of depth limit L

As with the breadth-first partitioning strategy used by PrologPF, the depth limit L determines the amount of computation done in the initial sequential phase of execution, and the number of open branches found.

An optimal value of the depth limit will minimise the amount of recomputation performed beneath the depth limit while discovering enough ports to provide sufficiently fine granularity of work in the subtrees to balance the assigned workloads.

The related issue with the BFP strategy is discussed in detail in Chapter 3, Section 3.6. The use of `kappa` avoids the requirement for oracles by interleaving the subtree search with the pseudo-breadth-first search phase. Potential optimisations using knowledge of the overall count or distribution of ports are not available with the use of `kappa`.

For some small values of the depth limit L, only a small number of ports will be discovered, perhaps smaller than the number of available path processors. In this situation the number of ports present at L will place an upper bound on the possible speedup. As with the breadth-first partitioning strategy, an improvement in speedup may be obtained by iterating through several increasing values of L until the number of ports discovered is at least some multiple of the number of available path processors.

An executable binary compiled using either PrologPF or with the `kappa` primitive can report the number of open oracles or ports found without further search if:

1. The program reports the number of open oracles or ports found on completion.

2. The program is assigned a unique processor number higher than the maximum open oracle or port count. The modulo arithmetic function used to select would in this case assign no ports or oracles to the selected path processor.

This technique suggests that optimising the number of open oracles or ports found S provides an estimate for a reasonable value of the depth limit L.

7.6.3 Oracle data structures imply limit on L

PrologPF with oracle-based breadth-first partitioning proceeds in two phases, building the *oracle stack* during the initial oracle discovery phase. For a given depth threshold L a number S of open oracles will be found, to be recorded in the oracle stack. PrologPF represents an oracle with a list of integers stored in an array, and the oracle stack is a two-dimensional array. The earlier implementation of DelphiKS used the number of clauses in each procedure to compress the clause index into a binary number of a variable number of bits [49]. A tree representation would also be more compact than the PrologPF arrays.

The storage of the oracles during the first phase of execution in PrologPF imposes a limit on the number of oracles that can be utilised. This places constraints on the acceptable values for the depth limit L. Figure 7.5 shows the increase in the number of open oracles discovered by PrologPF at increasing depth limits in the Pentominoes problem.

The oracles recorded on the oracle stack will each have a length equal to the selected depth limit. The oracle stack storage requirements for the Pentominoes program are given in the graph of Figure 7.6.

While the storage requirement of the oracle stack imposes a limit on the possible values of L, the use of the parallelisation primitive `kappa` has no accumulated oracle information as the execution proceeds. Thus the use of `kappa` is more tolerant of large values of L.

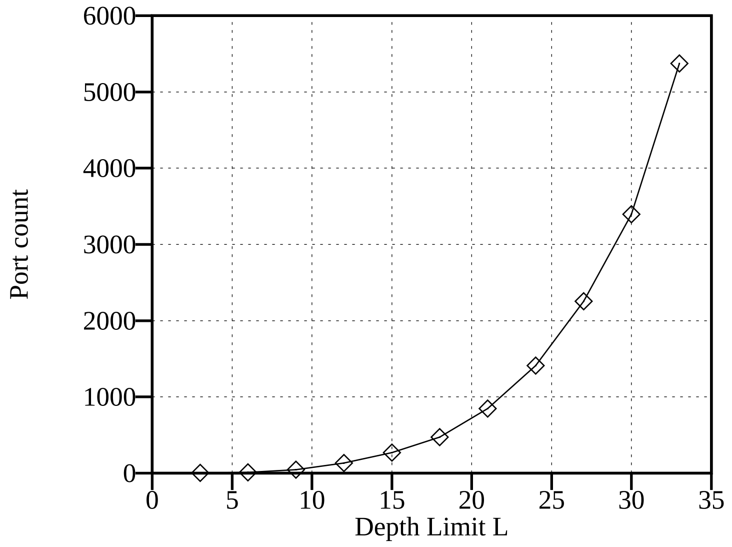


Figure 7.5: Count of open oracles for Pentominoes problem for $L = 3 \dots 33$.

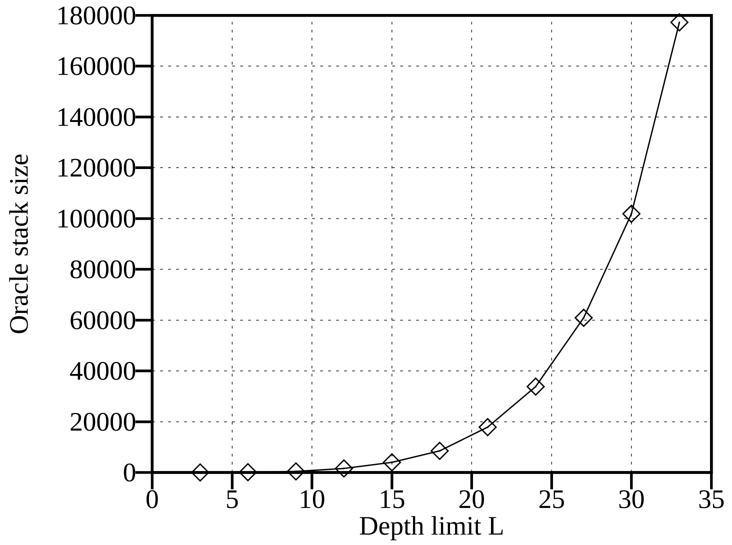


Figure 7.6: Oracle stack size for Pentominoes problem for $L = 3 \dots 33$.

7.6.4 Total `kappa` port count versus PrologPF open oracle count

The oracle-based partitioning algorithm used by PrologPF discovers all the open oracles at the selected depth limit *before* the assignment of the associated subtrees for search by the available path processors. As a minimum, the count S of open oracles is known before the allocation.

While PrologPF takes no advantage of this information in its simple fixed allocation algorithm, the information may be useful for future improvements.

The simple use of `kappa` searches each discovered subtree as execution proceeds in one phase, such that the total port count (equivalent to the PrologPF open oracle count) cannot be known as a given subtree is searched.

If the total port count is deemed essential for a worthwhile improvement in the port selection algorithm then the execution of a program using `kappa` can proceed in two phases. The first phase would be completely limited to the selected depth limit, i.e. the gating proposition would be `false`. This phase would return the number of ports found, and the count or other information made available to the port selection function as execution proceeds as before. This technique would double the overhead of the sequential component of the program execution. The times taken for execution of the sequential component of the Pentominoes problem for the range of depth limits used in Figure 7.5 are given in Figure 7.7.

The single-cpu execution time of the Pentominoes problem is 445 seconds, so the sequential execution time is reasonable, particularly for systems with few available path processors.

7.6.5 Work reassignment on path processor failure

In the event of path processor failure using PrologPF, the work can be passed to an alternative path processor by redistributing the affected oracles. The newly assigned path processors can efficiently recreate the environment needed at each subtree to repeat the search of the failed processor.

The impact of a failed processor may be greater with the simple use of the `kappa` primitive, as the reconstruction of the environment at a given *port* requires the complete search of the depth-limited subtree to the left of that port. This is because the port number is obtained from a count of the number of times the selected depth limit has been reached in the search so far, and the search is strictly depth-first, left-to-right.

The overhead of the processing required to reconstruct the environment at

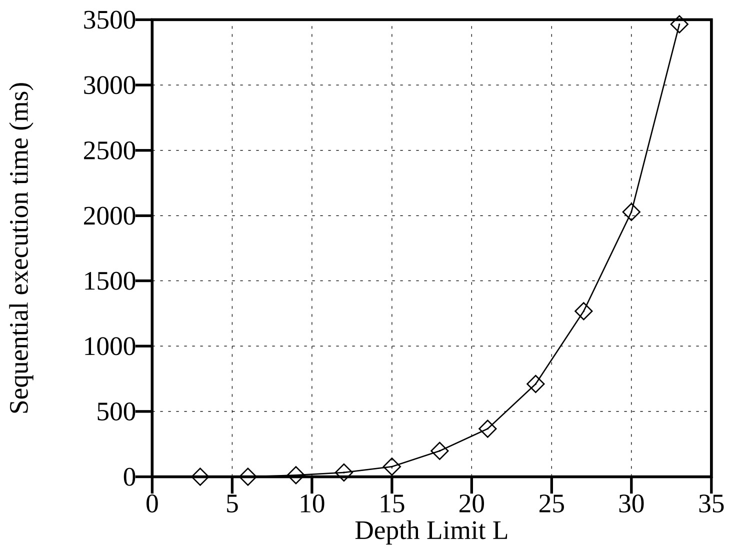


Figure 7.7: Sequential execution component of Pentominoes problem for $L = 3 \ldots 33$.

a given port is bounded by the sequential execution time of the problem up to the assigned depth limit. The sequential execution times for the Pentominoes problem at various depth limits is given in Figure 7.7.

7.6.6 Solutions found within the depth limit

For problems with multiple solutions distributed at various depths in the AND-OR search tree, it is possible to select a depth limit which divides the solutions into those beneath the depth limit and those in the subtrees beneath the ports. The search of the tree beneath the depth limit is repeated by all the path processors, and solutions in this part of the tree will thus be found and reported by all the path processors.

If the problem requires that duplicate solutions must be avoided, then the issue caused by the initial search repeated by all the path processors must be addressed. One technique to ensure unique solutions is to:

1. The path processor tags each solution with the *depth* at which that solution was found.

2. The control processor can accept solutions tagged with a depth less than that of the depth limit only from one path processor. Solutions

returned by other path processors with depths less than the depth limit are discarded.

As the depth limit is known to all the path processors and the control processor, then a similar technique could be used to ensure that only one path processor (for example, the path processor assigned the unique processor number 0) would return solutions below the depth limit to the control processor.

7.7 Repeated partitioning

As with the oracle based breadth-first partitioning used in PrologPF, the use of `kappa` can allocate differing workloads to the path processors, such that some will complete before others and become idle for the remainder of the computation. The issue for PrologPF is discussed in detail in Chapter 3, Section 3.6.

If many path processors have completed and become idle, but one or a few are still busy, then it may be worthwhile to redistribute the work from the busy processors to the idle processors. one mechanism to effect this is the repeated application of a depth limit within the subtree of a busy processor. The situation is pictorially represented in Figure 7.8.

If it is assumed that other path processors have completed the search of the subtrees beneath ports A, B and D in Figure 7.8, then path processor N assigned to port C is still executing. With a suitable extension of the port selection function, the group of path processors can all be set to search the subtree beneath port C at depth L, with a new depth limit for `kappa` of L'. The usual partitioning can occur at this new depth limit, such that the work has been assigned to all the previously idle path processors.

A given port in the search tree is uniquely identified by a list of (depth, port number) tuples. If `kappa` is associated with every relation in the user program, such that the depth used for partitioning is equal to the search depth in the AND-OR tree, then the list of (depth, port number) tuples is identical to the equivalent oracle used by PrologPF with an assumed depth increment of 1. In Figure 7.8 the sequence $[(L,C), (L',C1)]$ can be considered a compressed form of the oracle leading to port C1.

This repeated use of incremental depth limits with `kappa` is similar to the approach of *work splitting* discussed in Chapter 3, Section 3.6.5 for PrologPF. However, the overhead in following the list of (depth, port number) tuples is much greater then the efficient traversal of an equivalent oracle. For repeated use of depth limits with `kappa`, the left-most portion of each subtree

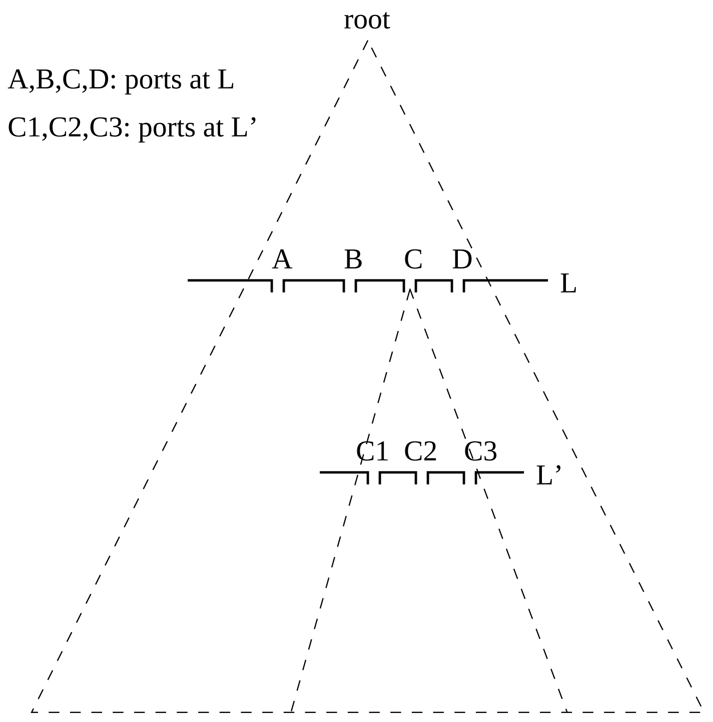


Figure 7.8: Repeated use of a depth limit and `kappa` within a subtree.

at each port in the list must be searched to discover the selected port at the next depth. For large balanced problems with a high degree of OR-parallelism and thus a large number of even subtrees at each selected depth, this overhead may still be acceptable.

An efficient implementation of a repeated partitioning strategy using both `kappa` and oracles is given in Chapter 8.

7.8 Conclusions

The fixed-allocation breadth-first partitioning algorithm with oracles used by PrologPF can be implemented without using oracles with a special parallel processing primitive called `kappa`. The efficient implementation and use of `kappa` requires support for user C programming and access to global C values in the Prolog system, but requires no modifications to the Prolog compiler or runtime system.

The use of `kappa` facilitates simple breadth-first partitioning of Prolog programs with similar performance characteristics and issues as those found with the oracle-based one-time breadth-first partitioning strategy tested with PrologPF.

Enhancements to the scheduling strategy possible through the availability of oracles in the PrologPF environment are not transferable to the use of `kappa` without oracles. Chapter 8 describes a strategy combining the advantages of `kappa` and oracles.

7.9 Summary

The implementation of the breadth-first partitioning scheduling strategy used by PrologPF uses oracles to represent the subtrees still to be explored at the open branches at the selected depth limit.

The BFP strategy proceeds in two distinct phases:

1. Oracle discovery, in which an oracle stack is built representing all the open branches at the selected depth limit.
2. Subtree selection and search, in which path processors are allocated disjoint subsets of the open oracles and search the referenced subtrees.

If these two phases are interleaved, a similar allocation strategy can be obtained without the use of oracles. The open oracles discovered in BFP are followed in the second phase to reconstruct the environment pertinent to the subsequent search of the dependent subtree. Without oracles, the environment constructed at the point of discovery of the open branch at the depth limit can be exploited if the subtree is searched immediately. The open branches at the depth limit are called *ports*, and a selection function similar to the oracle allocation function is required in each path processor, such that each processor skips over the ports allocated to other path processors.

The process of maintaining a global depth value and performing a port selection function has been integrated into a novel primitive called `kappa`. Calls to the special proposition `kappa` are embedded into the conjunctive subgoals of the user program, and the proposition has the following characteristics:

1. At depths other then the selected depth limit, `kappa` is transparent to the logic of the program, that is it always succeeds.
2. At the depth limit, `kappa` will succeed at ports allocated to the local path processor, and fail otherwise.

If the special proposition `kappa` is inserted before every subgoal in every clause, then the depth value maintained is the same as the depth into the transformed problem OR-only tree used by PrologPF [66]. `kappa` can be used selectively in the user program, as a minimum associated with just one user relation. The depth information is then limited to the depth of the nested calls of the selected relations.

The optimal distributed execution of programs using `kappa` is influenced by a number of factors:

1. The search of the problem tree at depths less than the selected depth limit is repeated in every path processor, and is effectively a sequential component of the execution. The time taken for this sequential component places an upper bound on the speedup available.

2. As with the one-time breadth-first partitioning strategy using oracles (BFP), the simple use of `kappa` relies upon a reasonable value for the depth limit at which partitioning takes place. A depth limit which is too small will not generate sufficient ports to permit an even distribution of the work. A depth limit which is too high will cause a high proportion of the available work to be executed sequentially, reducing the parallel speedup.

3. Unlike BFP, the use of `kappa` imposes no requirements for storage of oracle information as the depth-limited search progresses. In general, the use of `kappa` will accommodate larger values of the depth limit than BFP.

4. As the subtree search is interleaved with the port discovery process, the port allocation algorithm cannot use the knowledge of the total number of ports for any potential improvement in the efficiency of allocation. The equivalent information is available to BFP on completion of the initial oracle discovery phase.

5. The work in the subtree beneath each port may vary widely between different ports and between different path processors. The use of oracles in BFP provides the potential for the efficient redistribution of work from a busy processor to idle path processors. Redistribution of work with the use of `kappa` without oracles will incur a greater overhead of the recomputation of the subtree to the left of the current port.

The special proposition `kappa` provides the benefits of the one-time breadth-first partitioning strategy without the use of oracles. More complex partitioning strategies, such as work-splitting, are effectively supported by the combined use of `kappa` and oracles. Chapter 8 describes an implementation of work splitting with oracles and `kappa`.

Chapter 8

SOK: Splitting with Oracles and Kappa

This chapter describes the combined use of oracles and kappa. Both techniques are used in the recursive reallocation of work from busy to idle path processors, referred to as *work splitting.* The resulting scheduling technique improves upon the one-time allocation of work used in breadth-first partitioning by delivering greater speedup and removing the requirement for accurate selection of a depth limit parameter.

8.1 Background

The use of oracles in the breadth-first partitioning one-time scheduling algorithm is discussed in depth in Chapter 3. The alternative parallelisation primitive `kappa` is covered in Chapter 7. This section highlights the attributes of the two approaches exploited in a combined technique to provide effective work splitting.

8.1.1 Oracles

In the one-time partitioning provided by the breadth-first partitioning strategy described in Chapter 3 and [66], oracles are used to define subtrees for search by assigned path processors. Each oracle is followed to arrive at the root of the defined subtree, and the depth-first left-to-right execution strategy of standard Prolog is used within the subtree. The breadth-first partitioning strategy generates a complete set of oracles referring to every subtree with its root at the selected partitioning depth, and the oracles are allocated to the available path processors such that all the subtrees are searched.

The *current oracle* within a path processor is the sequence of clause indexes leading to the node in the search tree representing the current point in the depth-first left-to-right search. An *open* oracle leads to a choice point with further branches leading deeper into the search tree. Generally, the oracles issued by the depth-first partitioning strategy will be open oracles.

Associated with the scheduling strategy is the concept of *poisoned* oracles. If the workload is unevenly balanced among the oracles at the selected depth limit, one or more open oracles may lead to huge subtrees. Without work splitting, the long runtime of the path processors assigned to those oracles will dominate the overall runtime and reduce the parallel speedup. The assignment of an oracle referring to a very small subtree also reduces the efficiency of the parallelisation technique, as the path processor will perform the redundant computation involved in receiving and following the oracle without then performing much useful work. The breadth-first partitioning strategy mitigates the problem of poisoned oracles by requiring a partitioning depth at which many open oracles will be generated, such that the composite workload assigned to each path processor benefits from averaging.

Oracles have the following useful properties:

1. An oracle uniquely identifies a node within the search tree. Duplicate solutions can be recognised from their identical oracles. With the one-time assignment of work in the depth-first partitioning strategy, duplicate solutions can be found if they appear beneath the selected depth limit. In this case duplicates can be avoided with the simple mechanism of limiting those solutions to the path processor given a unique processor number $N = 0$. More complex strategies can provide speculative assignment of work in the knowledge that duplicate solutions can be recognised.

2. An open oracle can be treated as a reference to its underlying subtree. Another processor can use the oracle to recreate the environment at the root of that subtree, and can then independently perform the search of that subtree.

3. With each path processor following a strict depth-first left-to-right search strategy, an oracle can be considered to divide the search tree into two parts, to the left and right of that oracle respectively. A busy processor can return the oracle referring to its current node in the search tree. The implied left subtree represents the part of the tree already searched, while the right subtree represents the part of the tree still to be searched.

8.1.2 Kappa

The partitioning primitive `kappa` is described fully in Chapter 7.

The breadth-first partitioning strategy generates all the open oracles at a selected depth in the search tree, and then distributes all the oracles to the available path processors. The path processors then follow each assigned oracle to search each associated subtree. To reduce the communications requirements, *all* the oracles can be generated locally at every path processor, and each can use the allocation algorithm to select those for local search.

To reduce the overhead of processing the many oracles referring to small subtrees, the optimal partitioning depth limit will be that at which the number of open oracles S considerably exceeds the number of processors in the group G. Each path processor will be allocated S/G oracles. For example, at the optimal partitioning depth $L = 21$ for the pentominoes problem, 848 open oracles are discovered for allocation to the 30 path processors, so they each receive 28 or 29 oracles.

The parallelisation primitive `kappa` provides the same distributed behaviour as the allocation of the open oracles in the breadth-first partitioning strategy, without the requirement to accumulate and store the potentially large number of open oracles. The bounded-depth phase of BFP is interleaved with the search of the assigned subtrees as each node which would otherwise have generated an open oracle is discovered. The recomputation of the path up to the root of the selected subtree is avoided.

8.2 Work splitting

The combined support for both oracles and `kappa` provides an effective means to interrupt the work of a busy path processor and assign the remaining work to a newly formed group of idle path processors. The general idea is illustrated in Figure 8.1, in which one path processor is dividing its work among three others.

This section describes how the parallelisation support has been extended to facilitate work splitting, and discusses consequent scheduling issues. Effective work splitting is dependent upon:

- The ability of a busy path processor to efficiently communicate a specification of its remaining work to idle path processors.
- The reduction of the remaining work into a number of reasonably balanced subtasks.

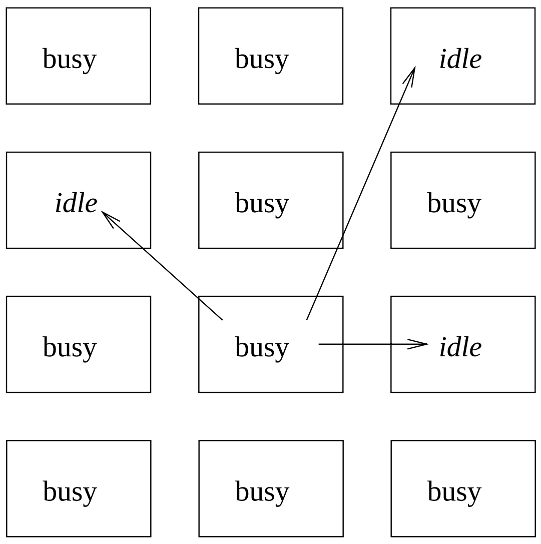


Figure 8.1: Work splitting.

- The ability of each assigned processor to recreate efficiently the context of the allocated subtask such that the work of the interrupted processor can be continued and ultimately completed.

Oracles provide effective support for the first and third requirements, while breadth-first partitioning with `kappa` is sufficient for the second.

8.2.1 At the busy path processor

Each path processor maintains the current oracle referring to its current node in the search tree. On interruption, the busy path processor communicates its current oracle and aborts the search of the current subtree. Assuming the busy path processor has been assigned a current partitioning depth limit L, the processor continues its search with the next allocated subtree at L. The point at which work splitting is initiated is described (and implemented) as an 'interruption'. This is to support scheduling strategies in which the interrupt is generated externally in addition to strategies in which some internal threshold (such as accumulated choice point count) triggers the work splitting in the busy path processor.

The situation at the busy path processor is illustrated in Figure 8.2. In the prototype implementation of the splitting with oracles and kappa (SOK)

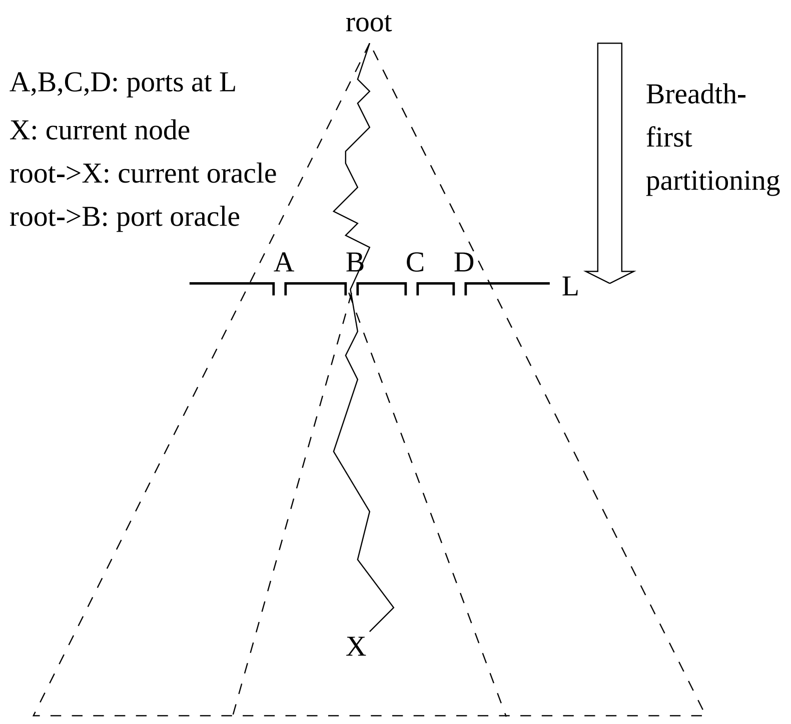


Figure 8.2: Interruption of a busy path processor.

strategy with PrologPF the interruption of the busy path processor causes it to:

1. Communicate its current oracle to the control processor.

2. Reset its state to the root of the search tree.

3. Continue the partitioning of the search tree at the depth limit L, but searching to the right of the previous current oracle. Due to the depth limit L, the busy path processor need only use the first L indexes of the oracle to determine the left bound of the continued search. This oracle prefix of length L is called the *port oracle*.

4. It is possible to interrupt the busy path processor when its current depth is below the depth limit L, in which case the response to the interrupt is deferred until the path processor reaches the next port at L.

8.2.2 At the idle path processors

A number of idle path processors are formed into a group with a new group count G', and new unique processor numbers $N' = 0 \ldots G' - 1$. They are given the current oracle from the interrupted busy path processor with a new depth limit L'. The situation is illustrated in Figure 8.3.

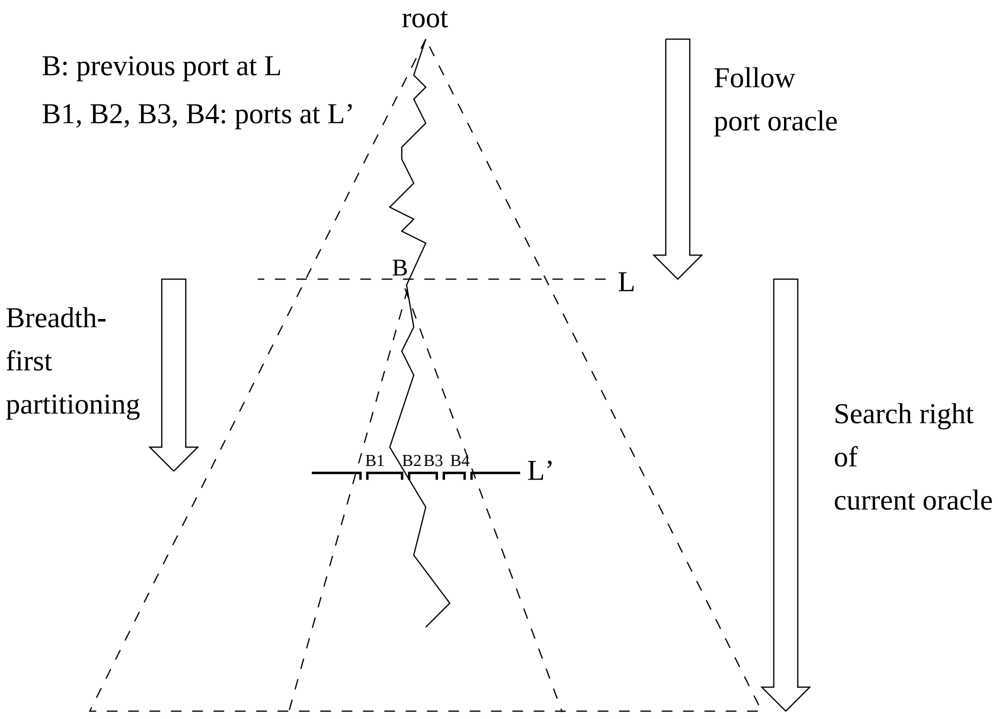

root
B: previous port at L
B1, B2, B3, B4: ports at L'
Follow port oracle
B
L
Breadth-first partitioning
B1
B2
B3
B4
L'
Search right of current oracle

Figure 8.3: Assignment of work to an idle path processor.

On receiving the oracle and the parameters G', N' and L and L', the path processor will:

1. Follow the oracle to a depth L to arrive at the root of the subtree previously partially searched by the interupted busy path processor.
2. Use the breadth-first partitioning technique with `kappa` to arrive at each allocated port at the new depth limit L' and search the defined subtree. Throughout this phase, the path processor's search is constrained to the right of the remainder of the provided oracle.

Once executing, the previously idle path processors become busy. If interrupted, a path processor in the new group will return its current oracle and depth limit L' and the splitting algorithm will be recursively applied.

8.3 Scheduling

As described above, PrologPF programs with support for splitting with oracles and kappa enable any busy path processor to be interrupted and the work divided between any number of idle path processors. These capabilities provide a foundation for a diverse range of scheduling algorithms. Choices to be made in the scheduling algorithm include:

- The criteria to determine *when* one or more busy path processors should be interrupted to redistribute their remaining work.
- How to determine *which* busy path processor to interrupt.
- The size of the new group to receive the divided workload.
- The specification of the incremental depth limit L' for the new group.
- Whether the scheduling decisions should be made locally within the busy or idle path processors, or whether more effective scheduling can be provided with a control processor.

The importance of efficient scheduling has been recognised in other OR-parallel Prolog implementations, such as Aurora [11] and Muse [5]. Butler and others discuss the issues of scheduling on the ANL-WAM OR-parallel system in [18]. The implementation of the SOK strategy in PrologPF has not so far been used to investigate these choices in any depth. A trivial scheduling algorithm was embedded into the control processor running `skynet` (Appendix A.4), with the following characteristics for an initial group of 30 path processors:

- A busy path processor will be interrupted when the number of idle path processors is ≥ 3. This parameter of SOK is called `split_g`.
- The busy path processor selected for interruption will be that with a current partitioning depth nearest the root of the problem search tree. Interruptions occur round-robin for busy path processors at the same least partitioning depth.
- The work of the interrupted path processor is assigned to 3 previously idle path processors.

- Two techniques to arrive at the incremental depth limit were evaluated: *fixed* and *doubling*. In the former the recursive depth limit is incremented by a fixed amount on each splitting of the workload, and in the latter the incremental depth limit L' is always double the depth limit L of the interrupted busy processor.
- Scheduling is managed by a centralised control processor, which receives the completion messages and generates the interrupts. The interrupted processors communicate their current oracle to the control processor, which selects the idle processors for work assignment and dispatches the work.

The one-time partitioning of the BFP strategy is analagous to the SOK strategy with `split_g` $> G$, such that no splitting takes place.

8.4 Results

For the performance results of the one-time partitioning BFP strategy in Chapter 3, the cpu time of the path processors could be used to arrive at the overall runtime. This removed consideration of the load time of the processes and the communication time of the solutions. This simplification was acceptable for a strategy with no scheduling communication after the intial distribution of the problem. The strategy of splitting with oracles and kappa (SOK) involves repeated communication during the execution of the problem, such that overall real runtime is important in the assessment of the scheduling technique. The runtime for the SOK strategy is measured from the point at which all the path processors are loaded with the sample program to the point at which all path processors are idle.

From the benchmarks used to evaluate the BFP strategy, the Pentominoes problem has been used for comparison with SOK in this section. The SOK strategy is implemented with three variants of the embedded parallelisation primitive, illustrated in Figure 8.4:

1. **No oracle partitioning:** the interpretation of an oracle as dividing the search tree into two parts is not exploited either above or below the depth limit. After interruption, a busy path processor must restart its search from the left-most path in the search tree below the depth limit to arrive at the next allocated port to then search the referenced subtree. On receiving the oracle, an idle path processor must fully search the first allocated subtree possibly duplicating work within that subtree already performed by the interrupted path processor.
2. **Oracle partitioning to ports:** The oracles returned by an interrupted busy path processor are truncated to length L, such that they

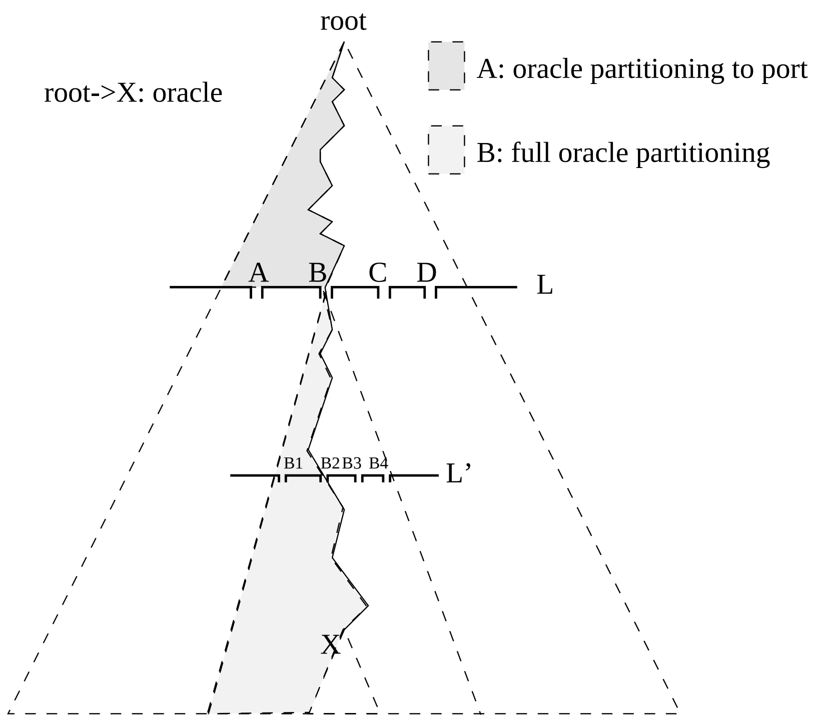


Figure 8.4: Interpretation of oracle as dividing search tree.

refer to the current port only. On interruption, the busy path processor has reset its state to the root of the search tree, but can efficiently continue its search by ensuring it searches to the right of the oracle leading to the port assigned to the idle processors. The busy path processor thus avoids duplicating the work performed in the shaded zone A in Figure 8.4, but the idle path processors will possibly duplicate work already performed in the subtree with its root at B. This partial use of the interpretation of the oracle as dividing the search tree provides a more efficient interrupt handling in the busy path processor, and provides a mechanism by which one or more of the idle path processors could efficiently divide the remaining work of the busy path processor at the depth limit L. The implementation tested here only divides the work of the busy processor within its current subtree.

3. **Full oracle partitioning:** The full current oracle is returned by a busy path processor on interruption ("root to X" in Figure 8.4). The busy path processor searches to the right of that oracle to arrive at the next allocated port, and the idle processors will search to the right of that oracle within the subtree, avoiding duplication of work in the shaded area "B" in Figure 8.4.

As real runtime has been used for the SOK speedup figures, the evaluation is more vulnerable to the load placed on the processors and network by other users. The results of several runs are averaged to produce the figures used in the speedup graphs. The variance was approximately 5%.

8.4.1 Fixed depth increment

Figure 8.5 shows the speedup performance for the SOK strategy with a fixed depth increment for a range of values of the initial partitioning depth limit $L = 1 \ldots 30$. The speedups for the BFP strategy is included for comparison. The fixed depth increment was chosen to be equal to the initial depth limit selected for the problem.

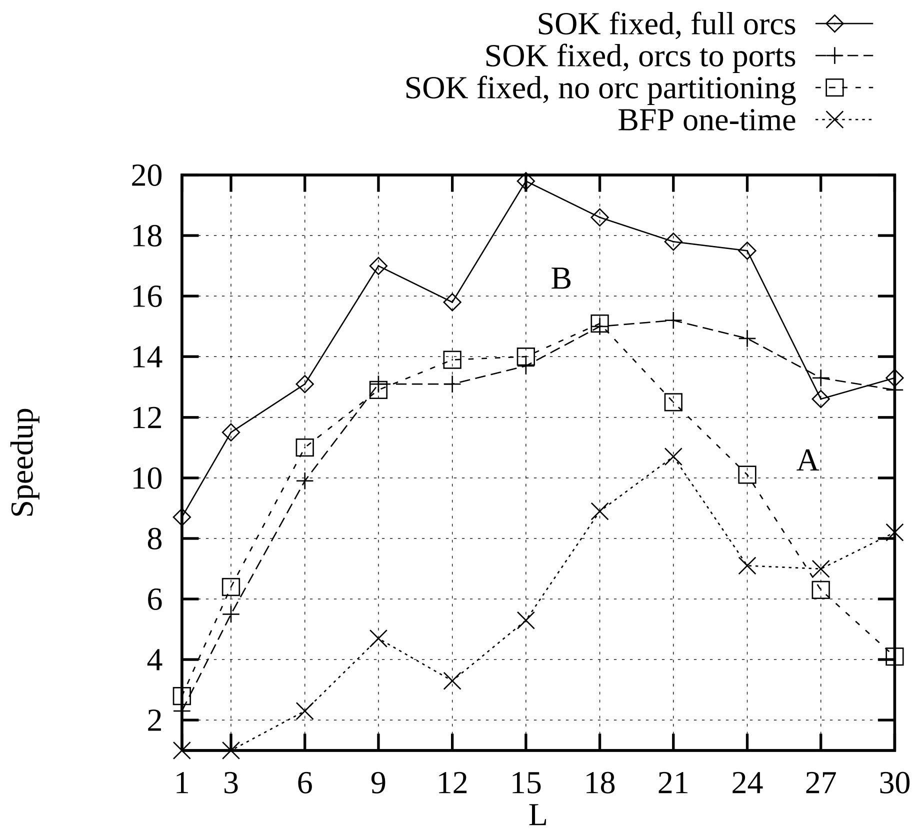


Figure 8.5: Pentominoes: splitting with fixed incremental depth limit.

The area labelled "A" in the graph in Figure 8.5 represents the improvement due to the more efficient interrupt handling provided in the busy path processor by exploiting the interpretation of the oracle as dividing the search tree beneath the depth limit L. Following the interruption the path processor does not then have to redo the work to the left of the oracle leading to the previous current port. The area labelled "B" in the graph in Figure 8.5 represents the improvement in efficiency resulting from the exploitation of the full oracle to avoid duplication of work within the assigned subtree.

The SOK strategy with a fixed incremental partitioning depth consistently outperforms the one-time partitioning of the BFP strategy. However, with no oracle partitioning, with the busy path processor resetting to the root of the search tree on interrupt, and with the simple scheduler implemented in skynet, the SOK strategy performs badly with large initial values of the partitioning depth parameter. For example, with an initial depth limit of 30, when 3 path processors have become idle a busy path processor will be interrupted, and the subtree defined by the returned oracle will be partitioned at a depth of 60. At this depth there is generally little work to do in the Pentominoes problem and those processors will quickly become idle, triggering the interruption of another busy path processor. As the problem nears completion, path processors become idle more rapidly than the remaining busy processors can respond to interrupts, and the system spends more time handling interruptions than performing useful work.

With small values for the initial partitioning depth, the use of the same parameter to provide the incremental partitioning depth results in inefficient partitioning as at each recursive step only a small number of ports are found at the new partitioning depth. The worst case is for $L = 1$ where splitting will occur at $L = 1$ and then $L = 2$ and $L = 3$ and so on. In effect, the system must split the work of a busy processor several times before an efficient partitioning is achieved.

8.4.2 Doubling depth increment

Figure 8.6 shows the speedup ratios achieved with the SOK strategy with the incremental depth limit recursively set to double each previous value. The speedups of the one-time partitioning strategy BFP are included for comparison.

The area labelled "A" in Figure 8.6 represents the improvement of the more efficient interrupt handling provided by the search to the right of the port oracle in the busy path processor. This improvement matches that found with the fixed depth limit incrementing technique. As with the fixed technique, the area labelled "B" shows the benefit gained from interpreting the full oracle as denoting the area of the search tree already searched.

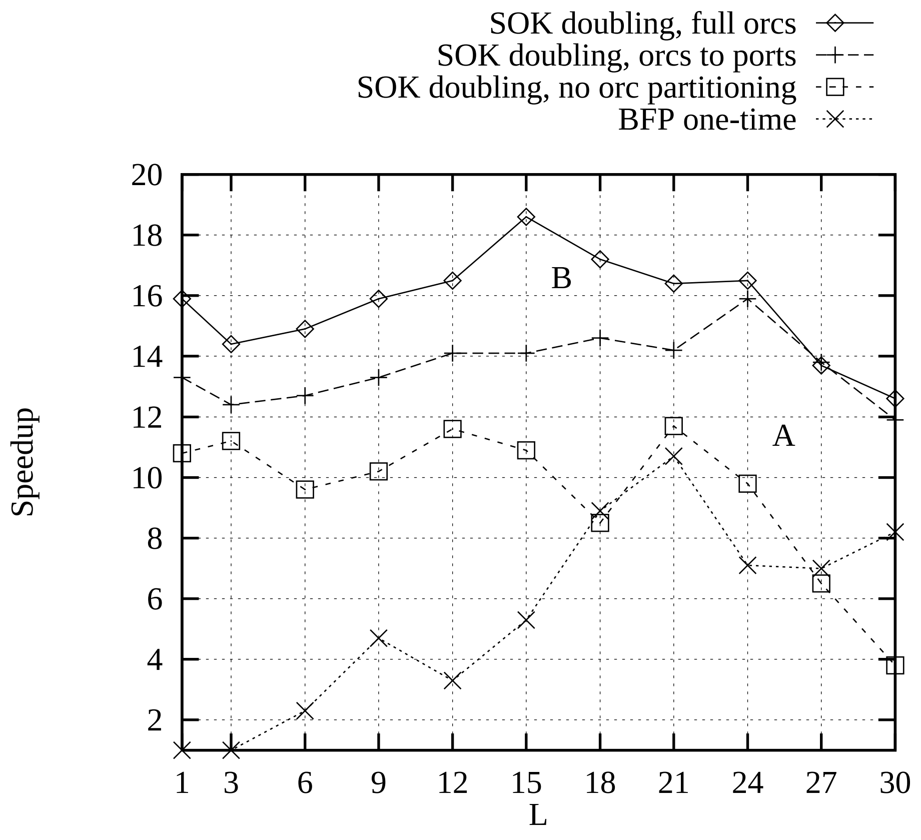


Figure 8.6: Pentominoes: splitting with depth limit doubling.

The technique of recursively using depth-limited search, with the depth limit doubling on each work assignment, was first suggested by Alshawi and Moran in [6]. The use of an initial depth limit of 1, and doubling this parameter on each splitting step, provides the most interesting behaviour. With the definition used for the initial query for the problem compiled with `prologpf`, the initial depth limit of 1 will result in only 1 port at that depth. Thus the program initially executes on one machine, but quickly causes all 30 machines in the test configuration to become busy through recursive splitting.

The graph in Figure 8.7 compares the performance of the fixed incremental depth technique with doubling. These results are from using the parallelisation primitive with support for the interpretation of the full oracle as a division of the search tree. The results for the two approaches are similar, but the doubling technique has the considerable advantage of effective speedup with an initial depth limit set to 1. The extended parallelisation primitive with the SOK protocol provides improved speedup over the one-time partitioning of BFP for all values of the initial depth limit L, and the SOK approach is much less dependent upon an optimal selection of L.

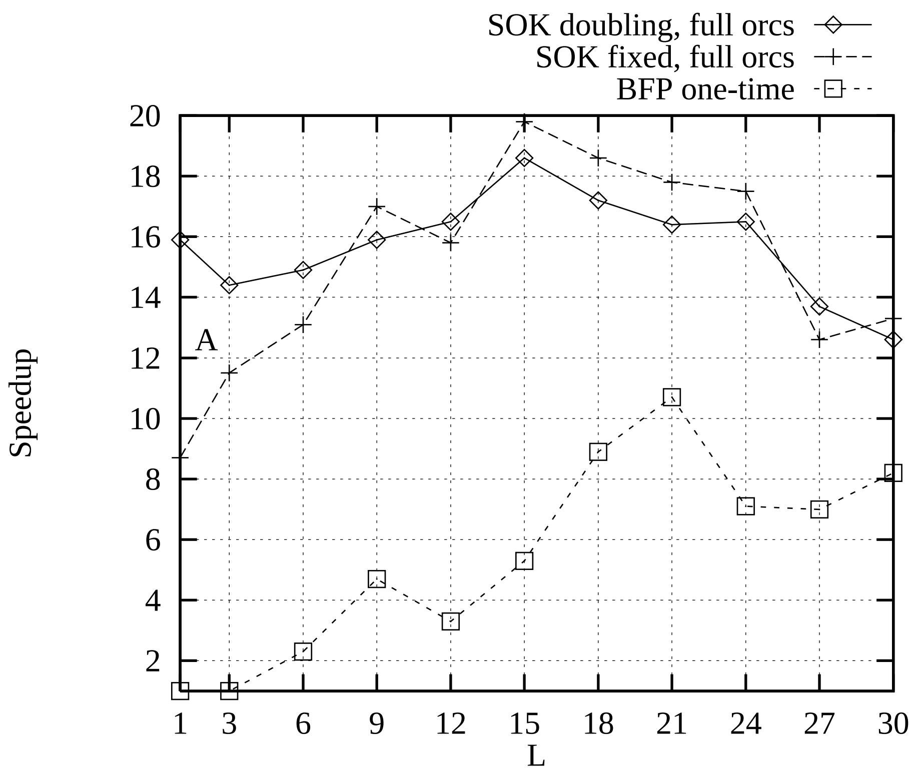


Figure 8.7: Pentominoes: fixed depth increment versus doubling.

The results show previously in this section compare the speedups achieved for a range of values of the initial depth limit L. Finally, the graph in Figure 8.8 compares the performance of the SOK strategy with doubling versus the one-time partitioning of BFP for a range of processor group sizes $G = 1 \ldots 30$. The SOK strategy with doubling provides greater speedup for all processor group sizes, does not have the performance variation of BFP.

8.5 Conclusions

The combined use of oracles to define each subtree for distributed search and `kappa` to provide partitioning leads to an effective parallelisation technique with better performance than can be achieved with one-time partitioning even with an optimal initial value of the BFP depth limit.

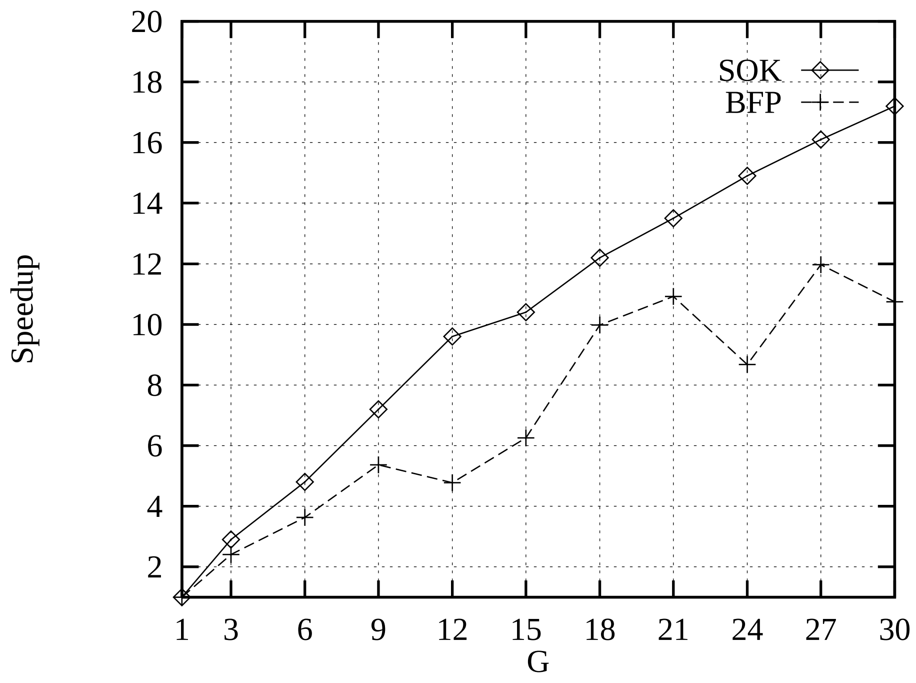


Figure 8.8: Pentominoes: full oracles and kappa versus one-time partitioning.

The extended capabilities of following an oracle and constraining the subtree search to the right of an oracle can be efficiently implemented with a simple primitive embedded in the user program[1].

The design of a system which permits any running processor to be interrupted and the workload efficiently split between a number of waiting idle processors provides a general platform for a variety of scheduling techniques. The trivial scheduling algorithm implemented in the control processor running `skynet` was sufficient to deliver the improved performance discussed in this chapter.

Recursive splitting of the workload, and the interpretation of oracles as dividing a tree into two parts provide an opportunity for the investigation of a new range of scheduling strategies.

[1]The code for the primitive as a 'C' macro is given in Appendix C.

8.6 Summary

Oracles can be used to:

- Uniquely identify a node or solution within the search tree.
- Define a subtree and associated context for search by another processor.
- Divide a subtree into two parts.

The *current oracle* defines the current position of a path processor within an assigned subtree. If an oracle leads to an intermediate node within the problem seach tree, it is called an *open oracle*. Scheduling strategies can suffer from the assignment of *poisoned oracles*, which can be those leading to huge subtrees or very small subtrees.

The partitioning primitive `kappa` described in Chapter 7 provides an effective means of dividing the search amongst multiple processors. The communication and recomputation overheads associated with the oracles providing an equivalent breadth-first partitioning strategy (BFP) are avoided by local traversal of the depth-limited subtree.

Support for oracles and `kappa` can be combined such that a path processor can follow an oracle to a certain depth, and then partition the workload of the subtree beyond that depth. If, on interruption, a busy path processor returns its current oracle, this support means that:

1. The current state of a busy path processor can be efficiently communicated to a control processor or directly to idle path processors.
2. A group of idle path processors, on receipt of the oracle, can quickly recreate the state of the interrupted processor and partition the remaining work across the new group.

The new approach described in this chapter addresses each of the following issues:

- **Small poisoned oracles:** fundamental to the use of `kappa` for breadth-first partitioning is the generation of a large number of ports at each incremental depth limit, such that a path processor will rapidly process the ports with small subtrees and move on without requiring further communication with a control processor or further interruption of busy processors.

- **Large poisoned oracles:** the interruption and splitting of the work of busy path processors means that the SOK technique is not vulnerable to the unequal distribution of work that affects BFP.

- **Selection of an appropriate depth limit:** the SOK strategy is effective with an initial depth limit of 1, such that initially only one path processor receives work with the others idle, and splitting repeatedly occurs until the work is allocated to all available path processors.

- **Low communications requirements:** to minimise the communication overhead the frequency of communication and the quantity of data transferred on each split must be kept to a minimum. Splitting with oracles and `kappa` reduces the frequency of communication by assigning multiple subtrees for search (at the incremental depth limit) on each assignment. The oracle and the parameters of the breadth-first partitioning phase provide a very compact means of communicating the work required.

- **Recovery from path processor failure:** the work assigned to a path processor is defined by the oracle and partitioning parameters. The information can be communicated to an alternative processor for the search to be repeated. Annotation of solutions with the associated current oracle provides a simple mechanism to avoid duplicates. The ease of recovery from processor failure using oracles extends the utility of the SOK strategy for large networks of general purpose workstations.

- **Control processor requirements:** the SOK strategy described above suggests the use of a control processor to initiate the work and provide global control for scheduling. The splitting technique described uses information local to the interrupted busy processor such that distributed or hierarchical control could equally be implemented, leaving the control processor to provide the user interface and startup and terminate execution.

The general support for work splitting and assignment permits a wide range of scheduling strategies. The simple strategy implemented for this evaluation interrupts the busy processor nearest the root of the search tree whenever a small number of path processors become idle. This simple scheduler was sufficient to produce significant improvement in parallel performance of the Pentominoes benchmark.

Chapter 9

Conclusions

Programs compiled with the PrologPF compiler can exploit the processing power available in a LAN or WAN of general purpose workstations, without requiring programmer annotations to the code to guide the parallelisation.

The breadth-first partitioning strategy minimises the communication necessary for distributed execution, such that the technique is optimised for systems with many processors but significant communications delay, such as the Internet. The combined use of oracles and the partitioning primitive `kappa` introduces communication during the execution to effect a dynamic redistribution of the workload. The use of oracles provides an efficient means of recovery from path processor failure, supporting the use of the technique in a widely distributed system. The single systems image provided by the combination of the PrologPF compiler and the skynet control system transform a generally idle network of workstations into a usable super-computer.

PrologPF provides effective speedup for large[1] problems containing sufficient OR-parallelism. For smaller problems, the runtime taken to generate sufficient oracles for distribution to the available path processors will dominate the overall runtime. Given a large problem, the speedup achieved by PrologPF is determined by the evenness of the balance of the workload assigned to the path processors.

The implementation of the oracle support in PrologPF imposes a 9% to 16% overhead on the execution of the compiled program on a single cpu. This overhead may be acceptable for the use of the compiled binaries in both a single-cpu and distributed environment, or the support for oracles can be more efficiently implemented in the virtual machine rather than the source transformation technique used with `prologpf`.

[1]In this context, *large* implies a single-cpu runtime greater than a few seconds.

The initial phase of the breadth-first partitioning strategy produces a number of open oracles at the selected depth. The distribution of work beneath these oracles can be quantified with the simulation of a subsequent execution of the problem with the number of path processors G set to the number of open oracles S discovered at the depth limit L in an earlier execution with $G = 1$. An analysis of some commonly available Prolog benchmark programs shows that many of those oracles will refer to small subtrees, with a sparse distribution of oracles referring to large subtrees representing most of the available work. To achieve effective partitioning of the search tree, the depth limit at which open oracles are found must generate a sufficient number of oracles referring to large subtrees, such that all path processors can expect to receive a number of these sizeable oracles. The general issue for oracle-based scheduling strategies such as the breadth-first partitioning is that of the *poisoned oracle*. For any strategy without subsequent work splitting, a poisoned oracle is likely to be an oracle which refers to a very large subtree, such that the uninterrupted sequential search of that subtree dominates the overall parallel runtime. For other strategies used with oracles requiring communication, such as the automatic partitioning strategy of DelphiKS, a poisoned oracle is that referring to a very small or nil subtree.

An effective work estimation function could lead to better balancing of the work assignment to the path processors. The generally sparse and random distribution of the large oracles means that work estimation based upon the arithmetic mean of the oracle neighbours will not succeed. Partially searching the subtree but limiting the number of choice-points traversed similarly relies upon a limited measure of workload beneath certain oracles being used as an estimate for others.

The breadth-first partitioning strategy tested with PrologPF succeeds because a large number of oracles can be efficiently generated and allocated, and the process of following an assigned oracle within a path processor is very efficient. The set of oracles assigned to a given path processor will typically include many referring to very small subtrees, but the within the set a few will refer to larger subtrees, providing sufficient work for effective partitioning.

The meta-logical predicate *cut*[2] used in sequential Prolog programs to prune the search tree does not have an efficient implementation in a distributed OR-parallel environment. The propagation of a *cut* in the search tree of one path processor requires the communication to the other path processors searching affected subtrees. A program containing a small procedure with *cut* may encounter that predicate thousands of times each second. PrologPF provides a useful programming system for logic programs that in Prolog

[2]Written "!" in Prolog.

would contain *cut* by supporting deterministic execution through the use of higher-order functional programming, explicit negation through the use of boolean functions, and permitting *cut* in deterministic relational procedures.

The implementation of higher-order functions in PrologPF has been achieved while maintaining upwards compatibility with standard Prolog. A consistent syntax is possible for the combined styles, permitting reduction of functional terms as arguments to relations and requiring relational goals as conditions in functions. Allowing the definition of functions and relations of the same name but different arities within the same PrologPF program facilitates the straightforward mapping of existing deterministic library relations into new functions. All function calls in PrologPF appear in arguments to relations. Function failure can be supported as an exception propagated to the outermost call in the argument term and treated as unification failure leading to the failure of the underlying relational subgoal.

PrologPF produces efficient compiled code, with the compiler built upon the `wamcc` Prolog to C compiler [30]. Efficient support for oracles can be provided though the use of simple primitives, `o_kbuild` and `o_kfollow`, automatically embedded in the user program by a pre-processing pass of the compiler. The definition of the primitives permits their inline implementation as C macros, and the source is given in Appendix C.

For the simple one-time breadth-first partitioning scheduling algorithm tested with PrologPF, an equivalent behaviour can be achieved through the use of a novel primitive `kappa`. The primitive, embedded as a proposition in the user program, performs dynamic pruning resulting in the same partitioning of the search as PrologPF but without the intermediate use of oracles. The absence of oracles means that basic refinements to the technique available to a future development of PrologPF may not be feasible with `kappa`. Implemented as a two-line C macro, `kappa` can be used implement the BFP strategy with most standard Prolog compilers.

The use of oracles can be combined with the partitioning primitive `kappa` to support scheduling strategies using the recursive reassignment of work from busy to idle path processors. While most OR-parallel Prolog systems assume a work assignment rate of many thousands of times per second, the partitioning techniques discussed in this dissertation are effective with work reassignment occurring orders of magnitude less frequently.

The use of oracles as the fundamental mechanism to communicate work for distributed execution provides an efficient means of recovery from processor failure. The work can be reassigned to an alternative processor without the other path processors in the group being affected and with a minimal impact on the total execution time.

9.1 Future work

The scheduling strategy used in PrologPF assumes the work beneath the discovered open oracles is randomly distributed, and no attempt is made to associate a work estimate with each oracle. If an effective estimation technique could be found, then the effectiveness of the one-time allocation of work would be greatly improved.

In PrologPF, the parallelisation of the user program is limited to the logical component. Functional reduction within a specific subgoal is executed sequentially. Further research may lead to an effective method for extending the use of oracles to parallelise functional execution.

PrologPF supports the use of *failure* as an uncatchable exception within a functional reduction, which is propagated to the top level function call and leads to failure of the underlying subgoal. More flexible support might be provided through the general provision of exception support in the functions, unified with the exception support of standard Prolog.

Appendix A

System description

A.1 System overview

The sofware components of the PrologPF system are:

- The PrologPF compiler
- The Skynet control processor
- The Skyhub group controller
- The `ppc` path processor control daemon
- The compiled user program

A diagram of the systems architecture is given in Figure A.1.

After compilation of the user program with the PrologPF compiler, the executable binary is made available to every path processor. A set of commands are provided on the control processor to establish communication between the control processor and the selected number of path processors, via the intermediate Skyhub processors. The control processor can then initiate and control the execution of the user program on every selected path processor, accumulate statistics, and display the collected results.

The Skynet system is general, such that its use is not limited to `prologpf` executables and any program can be launched on the distributed processors with the results (on 'standard output') returned to the control processor. The `prologpf` executables have the unique property that the same object program on each processor still results in each processor performing a fraction of the total work, with the sum of the parts producing the same results

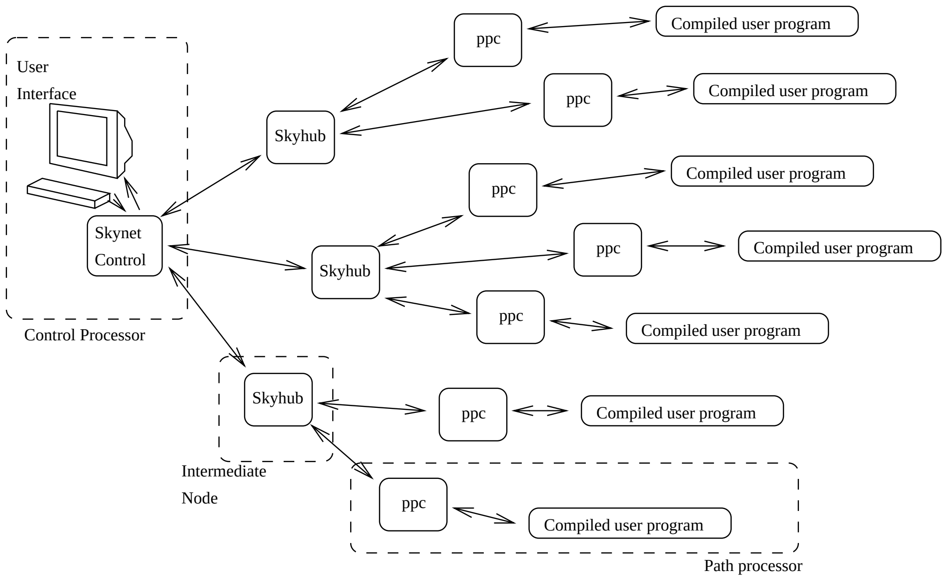


Figure A.1: System communications architecture

as the same object program executing on a single cpu. Non-`prologpf` executables run normally to completion on every machine, so nominally the same results are produced by all. However, Skynet provides some useful support for non-`prologpf` executables in the distributed system for host-specific programs such as `ps` to return the running processes and cpu utilisation or even `hostname` to return the list of hostnames.

A.2 The Prolog compiler: `wamcc`

PrologPF is built upon the excellent Prolog to C compiler `wamcc` developed by Daniel Diaz at INREA [30]. The `wamcc` is written in Prolog, as a series of modules named `wamcc0.pl` through to `wamcc8.pl`.

Starting with a user program `foo.pl`, the compilation with `wamcc` proceeds as follows:

```
wamcc -c foo.pl
gcc -c foo.c
gcc -s -o foo -lwamcc
```

The first compilation step creates two C source files, `foo.c` and `foo.usr`.

The main C file `foo.c` has definitions to include the file `foo.usr` and a number of header files with utility functions and macros. The file `foo.c` is valid C, but closely resembles WAM code [73], with each instruction defined as a C macro. Each Prolog procedure translates into a preamble and postamble wrapper of C code enveloping the C macros defining the WAM instructions.

After `wamcc` has produced the C code, this program can be compiled and linked with the standard system C compiler to produce an executable binary. Precompiled library functions are supplied via the library `libwamcc.a` referenced in the final linking step.

A.3 The PrologPF Compiler: `prologpf`

The PrologPF compiler extends `wamcc` with recognition of functional terms and the generation of the appropriate C code. The extension to the `wamcc` compiler is predominantly in the additional Prolog modules `wamcc_ocode`, `wamcc_kcode` and `wamcc_fcode`. Support in C for the management of oracles has been added to the `libwamcc.a` library. Compilation of a user PrologPF program is similar to the process with `wamcc`:

```
prologpf -ocode -fcode -ppf -c bah.pl
gcc -c bah.c
gcc -s -o bah -lwamcc
```

As with `wamcc`, the `prologpf` compilation step produces a C file for subsequent compilation with the system C compiler. The additional flags have the following meanings:

-ocode: Produce code with embedded 'C' oracle support suitable for distributed execution with the one-time partitioning BFP strategy described in Chapter 3.

-ocode_pl: As with `-ocode` except that Prolog procedures and data structures are used to provide the oracle support. This can increase runtimes by a factor of four, but provides a flexible development environment to experiment with a given problem.

-fcode: Recognise function definitions and functional argument terms and produce appropriate code.

-kcode: Produce code with embedded support for oracles and partitioning with `kappa`, supporting work splitting and reassignment as in the SOK strategy described in Chapter 8. `-ocode`, `-ocode_pl` and `-kcode` are mutually exclusive.

-ppf: Produce an intermediate `bah.ppf` file showing the additional embedded predicates for the distributed and functional support.

PrologPF programs compiled with the `-ocode` flag accept three additional command line arguments: the path processor group count G, the unique processor number N, and the partitioning depth limit L. For example, the command `bah 12 5 27` will execute the program `bah` for processor number `5` assumed to be within a group of `12` path processors, with a partitioning depth limit of `27`.

The `-kcode` flag produces an executable which produces a behaviour which is an extension of that for `-ocode`. If the command-line arguments G, N and L are specified the executable uses the one-time BFP strategy. Otherwise the executable implements the SOK strategy described in Chapter 8, with a toplevel which waits until the arguments of the SOK strategy are received (G, N, L, L', Oracle) and performs the specified search and on completion waits for further work. If interrupted, the executable will return its current oracle and continue.

If the `-ocode`, `-ocode_pl` and the `-kcode` flags are omitted, the executable binary executes normally on a single cpu and contains no oracle management overhead, and cannot be run on a distributed system. With the `-ocode` or `-kcode` flag set, the program is still suitable for standalone execution on a single cpu simply by specifying a path processor group count $G = 1$, unique processor number $N = 0$, and the depth limit should be $L = 1$, for example `bah 1 0 1`.

If the `-fcode` flag is omitted, the function definitions and function application terms are treated as standard Prolog facts and compound argument terms respectively, and no functional reduction support is included.

The command `prologpf -c bah.pl`, specifying no functional or distributed support, produces the same compilable C source as `wamcc -c bah.pl`.

A.4 The Network System: `skynet`, `skyhub` and `ppc`

The general approach to executing PrologPF programs in parallel on a distributed network of workstations is to launch the same compiled binary on each workstation, with the first argument G set to the common group size and the unique processor number N ranging from $0 \ldots G-1$. The user selected depth limit L is added as the third argument passed to every path processor.

The daemon `ppc` runs continuously on every path processor, waiting for commands over a TCP/IP socket connection with the control processor. The command `start_prog` instructs the daemon to fork an execution of the user program with the assigned parameters. `ppc` remains connected via Unix *pipes* to the user program while it executes, accepting statistics and results from the user process and forwarding them to the control processor. Other commands from the control processor instruct `ppc` to interrupt or terminate the execution of the user process.

The control processor executes the program `skynet`, which communicates with the `ppc` daemon on each path processor, coordinates the execution of the multiple copies of the user program, and provides an interface for the user. A typical display seen by the user is illustrated in Figure A.2.

The window labelled "Skynet Control" is the command-line interface to skynet, providing commands such as `sky_connect` to establish connection with a `ppc` daemon, and `sky_bfp` to initiate the breadth-first partitioning strategy on all the selected path processors.

The window labelled "Status" accumulates runtime information as the distributed execution of the user program progresses. In particular, the `running` field indicates the number of path processors currently executing the user program, and `completed` shows the number of path processors which have completed their search and become idle.

The "Solutions" window simply displays solutions as they are returned from the path processors. The "Incoming" window displays the complete log of all communications from all the path processors. In the example shown, the solution returned by the user program is the atom `found`, to reduce the volume of information accumulated in the "Solutions" window. The voluminous solution is recorded in the "Incoming" log.

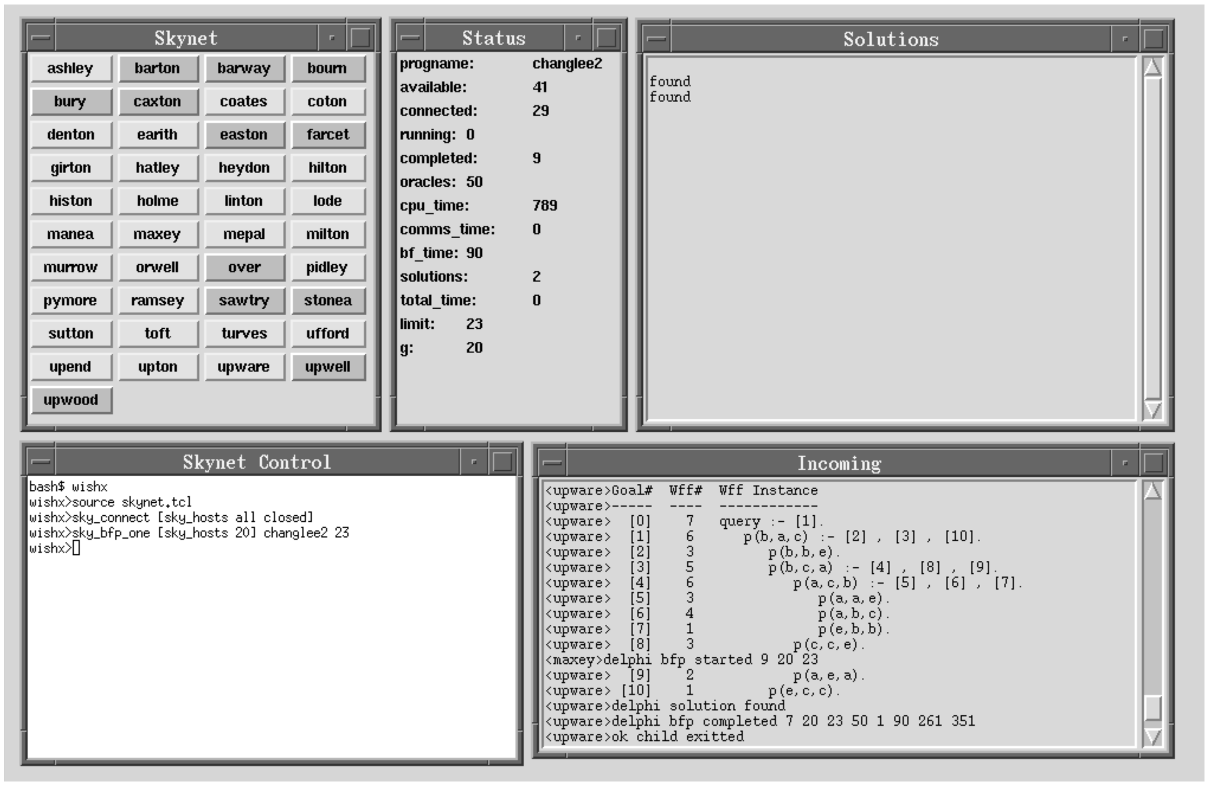
The "Skynet" window is a graphical display of the status of each of the available path processors. Each button is displayed in one of five colours:

Grey: Path processor not yet contacted. A user click on the button will cause `skynet` to connect to the path processor `ppc` and change the button to yellow, or red if the connection fails.

Yellow: `skynet` connected to `ppc`. A user click will disconnect `skynet` from the path processor `ppc` and change the button back to grey.

Blue: The user program implementing the SOK strategy is loaded and idle on the path processor. Programs executing the one-time partitioning BFP strategy never display as blue.

Green: Path processor is currently busy with user process. A user click will send a command to the path processor `ppc` to *kill* the user process.



```
progname:       changlee2
available:      41
connected:      29
running:  0
completed:      9
oracles:  50
cpu_time:       789
comms_time:     0
bf_time:  90
solutions:      2
total_time:     0
limit:    23
g:        20
```

```
found
found
```

```
bash$ wishx
wishx>source skynet.tcl
wishx>sky_connect [sky_hosts all closed]
wishx>sky_bfp_one [sky_hosts 20] changlee2 23
wishx>
```

```
<upware>Goal#  Wff#  Wff Instance
<upware>-----  ----  ------------
<upware>  [0]    7   query :- [1].
<upware>  [1]    6      p(b,a,c) :- [2] , [3] , [10].
<upware>  [2]    3         p(b,b,e).
<upware>  [3]    5         p(b,c,a) :- [4] , [8] , [9].
<upware>  [4]    6            p(a,c,b) :- [5] , [6] , [7].
<upware>  [5]    3               p(a,a,e).
<upware>  [6]    4               p(a,b,c).
<upware>  [7]    1               p(e,b,b).
<upware>  [8]    3            p(c,c,e).
<maxey>delphi bfp started 9 20 23
<upware>  [9]    2            p(a,e,a).
<upware> [10]    1         p(e,c,c).
<upware>delphi solution found
<upware>delphi bfp completed 7 20 23 50 1 90 261 351
<upware>ok child exitted
```

Figure A.2: Skynet control processor user interface.

A program using the SOK strategy will cause the button to change to blue on completion, then back to green when work is reassigned to the path processor. The button will change back to yellow when confirmation of the child exit is received.

Red: Path processor unavailable.

The BFP strategy is initiated with the command `sky_bfp` (for all solutions) or `sky_bfp_one` (for one solution), as in:

`sky_bfp` <*host_list*> <*prog_name*> L

The count of hosts in *host_list* provides the value of G for the strategy, and unique processor numbers $0 \ldots G-1$ are used for N. The SOK strategy implemented with `prologpf` uses an initial depth limit of 1 and doubling to provide the incremental depth limit, so the user need not specify L, and the strategy is initiated with the `sky_kappa` command:

`sky_kappa` <*host_list*> <*prog_name*>

When the user issues the `sky_bfp` command or the `sky_bfp_one` command in the "Skynet Control" window, the buttons of the selected path processors will change from yellow to green and the user processes start execution. A path processor running the SOK strategy which is interrupted causes the associated button in the "Skynet" window to briefly change its text to white. With the one-time BFP strategy, the buttons return to yellow as the path processors complete their assigned work and the user program is exitted. With the SOK strategy, the buttons turn blue as path processors complete each assigned piece of work, and back to green when executing a new workload. With the SOK strategy `skynet` will send a command to all user programs to exit when there is no further work to be executed, and the buttons will turn yellow together as each user program exits.

`sky_net` provides a facility to open a console to any host in the network, providing a filtered version of the "Incoming" window and a command interface to `ppc` such that the progress of an individual path processor can be monitored and controlled.

The process `skyhub` provides a transparent multiplexing function between `skynet` and the many `ppc` daemons. The primary requirement for the `skyhub` process was to overcome the 64 socket-connection limit of the DECstation 3100's used for the research, although in fact never more than 42 were available. The `skyhub` process has facilities for the local storage and manipulation of variables by the controlling `skynet`, for a possible future hierarchical implementation of a control system.

Appendix B

Benchmarks

This appendix gives the source code of the benchmark programs used to evaluate the distributed performance of PrologPF. Graphical representations of the search trees are included for each benchmark.

B.1 Queens

The Queens benchmark places N queens on an N by N chessboard, such that no pair of queens are on the same horizontal, vertical or diagonal.

```
:- main([f_utils,o_kutils]).

% get_solutions(N,S) succeeds if S is a solution for
%  the placement of N queens on an NxN chessboard.

get_solutions(Board_size, Soln) :- solve(Board_size, [], Soln).

% solve(N, Initial, Final) succeeds if Final is a solution
%  to the NxN queens problem and Final is an extension of
%  the partial solution in Initial.

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).
```

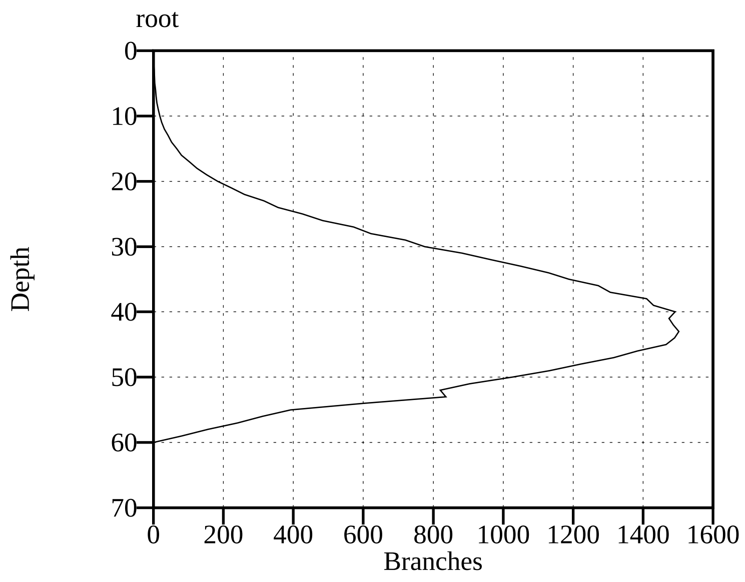


Figure B.1: 8 Queens: search tree.

```
% newsquare(Initial, Square, N) acts as a generator for
%  'safe' squares in the next column to those already
%  allocated in the partial solution in Initial.

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,N) acts as generator for X = N down to 1.

snint(X, X).
snint(N, NPlusOneOrMore) :- M is NPlusOneOrMore - 1, M > 0,
    snint(N,M).

% notthreatened(I,J,X,Y) succeeds if a queen on square (I,J)
%  does not attack square (X,Y).

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, !.
```

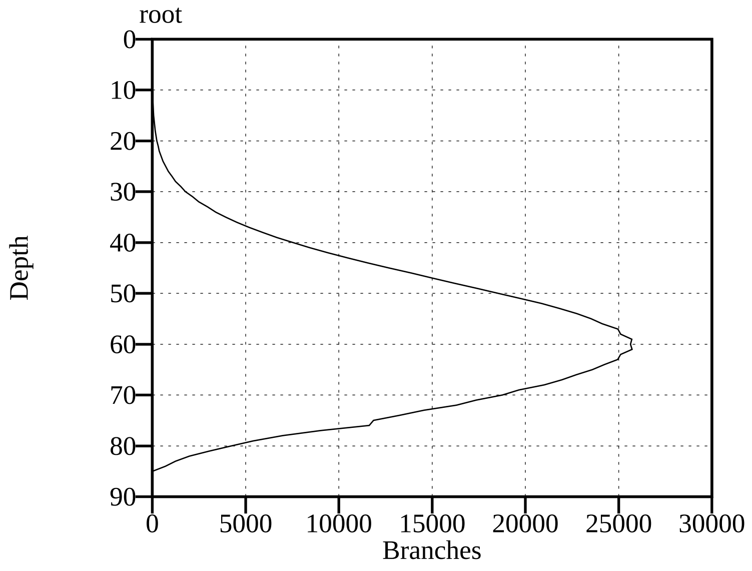


Figure B.2: 10 Queens: search tree.

```
% safe(X,Y,Initial) succeeds if square (X,Y) is not attacked
%  by any queens in the partial solution Initial.

safe(X, Y, []) :- !.
safe(X, Y, [square(I, J) | L]) :-
    notthreatened(I, J, X, Y), safe(X, Y, L).

% o_query is called by the PrologPF system to obtain the
%  solutions.

o_query :- get_solutions(8,X), o_ksoln(X).  % or 10 for 10-queens

% o_kloop is a top-level system predicate which responds to
%  the commands from the control processor.

:- o_kloop.
```

B.2 Pentominoes

The Pentominoes benchmark places twelve geometric shapes on a twenty by three board.

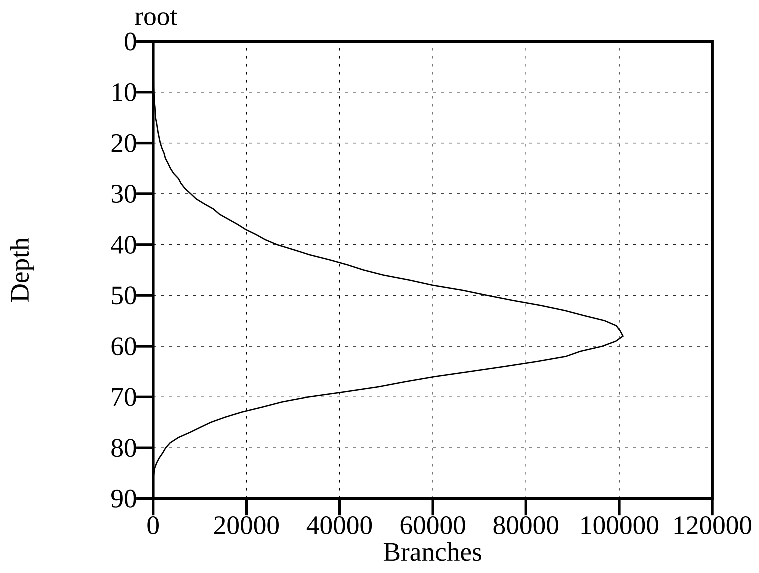


Figure B.3: Pentominoes: search tree.

```
:- main([f_utils,o_kutils]).

% solution(H) succeeds if H is a solution to the problem.

solution(H) :- initial_state(Si),
               can_reach(Si,Sf),
               final_state(Sf),
               Sf = state(_,_,H).

% initial_state(S) builds the term S representing the initial
%  state of the problem, using gen_board to build each
%  column of the 20x3 board.

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(S) succeeds if S indicates all pieces have been
%  placed.

final_state(state(_,[],_)) :- !.

% can_reach(S1,S2) succeeds if the second state of play
%  represented by S2 can be reached from state S1.

can_reach(S1,S2) :- trans(S1,S), S = S2.
can_reach(S1,S2) :- trans(S1,S), can_reach(S,S2).

% trans(S1,S2) succeeds if state S2 can be reached from
%  state S1 through the valid placement of one of the
%  remaining pieces.

trans(State,New_State) :-
    State = state(Board,Pieces,History),
    del(Piece,Pieces,New_Pieces),
    pent(Piece,Orientation,Pattern),
    play_pent(Board,Pattern,New_Board),
    New_State = state(New_Board,New_Pieces,
                      [[Piece,Orientation] | History]).

% del(X,L1,L2) succeeds if list L2 is equal to list
%  L1 with the removal of an element equal to X.  It
%  is used as a generator for elements of L1 (pieces).

del(X,[X|Y],Y).
del(X,[Y|Z], [Y|Z1]) :- del(X,Z,Z1).

% play_pent fits the pattern represented by a pentomino
%  of a given orientation onto the board, and generates
%  the remaining new board.

play_pent(Board,Pattern,New_Board) :-
    match(Board,Pattern,Board1),
    trim(Board1,New_Board), !.

% trim(B1, B2) succeeds if board B2 is the pattern
%  representing the remaining clear squares on the
%  board excluding the pieces and border of the board B1.

trim([],[]) :- !.
trim([border|T],Board) :- trim(T,Board).
trim([piece|T],Board) :- trim(T,Board).
trim(Board,Board) :- Board = [no_piece|_].

% match succeeds of the pattern representing a selected
%  piece can be placed on the board.

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).


% the following terms represent the patterns of all 12
%  pentominoes in each orientation.

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]).

query :- solution(X), o_ksoln(X).

:- o_kloop.
```

B.3 Prolog Technology Theorem Prover

The Prolog Technology Theorem Prover is a Prolog program of approximately 1500 lines. Sample problems are provided as Prolog compound terms, and passed to PTTP for translation into sound Prolog programs for compilation and execution. The source for PTTP, developed by Mark Stickel [70], is the commercial property of SRI[1]. The sample problems, taken from Chang and Lee [21] and Overbeek [37], are listed below.

B.3.1 Chang and Lee Example 2

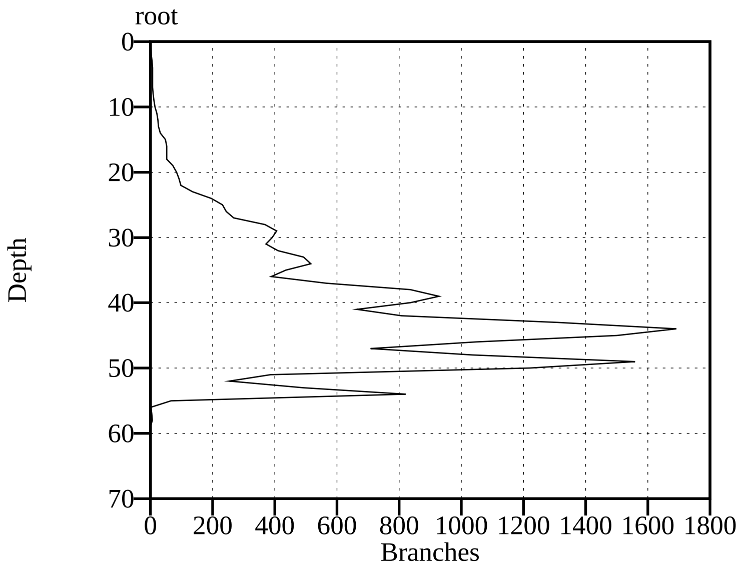


Figure B.4: Chang and Lee Example 2: search tree.

```
p(e,X,X)
p(X,e,X)
p(X,X,e)
p(a,b,c)
p(U,Z,W) :- p(X,Y,U) , p(Y,Z,V) , p(X,V,W)
p(X,V,W) :- p(X,Y,U) , p(Y,Z,V) , p(U,Z,W)
(query :- p(b,a,c))
```

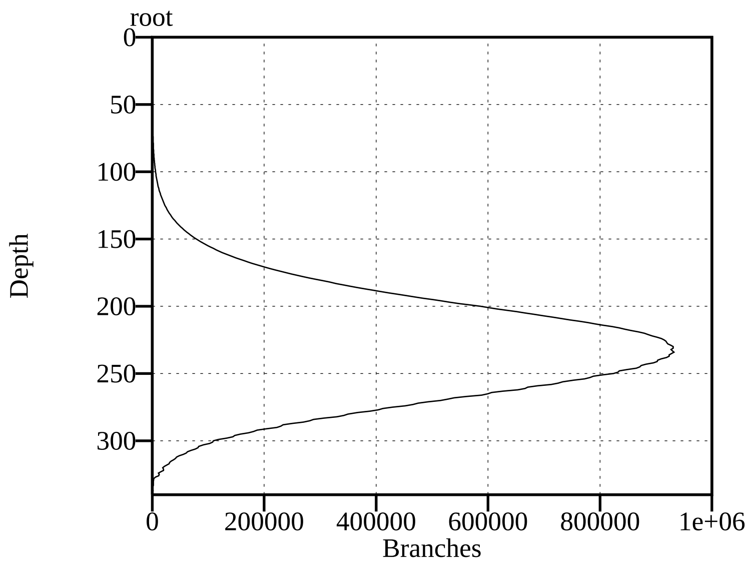


Figure B.5: Overbeek Example 4: search tree.

B.3.2 Overbeek Example 4

```
p(e(X,e(e(Y,e(Z,X)),e(Z,Y))))
p(Y) :- p(e(X,Y)), p(X)
query :- p(e(e(e(a,e(b,c)),c),e(b,a)))
```

[1]Artificial Intelligence Center, SRI International, Menlo Park, California 94025.

Appendix C

Source code for the parallelisation primitive

This appendix gives the source code for the simple parallelisation primitive supporting oracles and kappa.

To embed the parallelisation support, each user clause is modified to include a special goal `o_kbuild(N)` where `N` is the index of the clause in the current procedure. This example program is transformed as follows:

```
a(X) :- b(X), c(X).
a(X) :- d(X).
a(z).

b(X) :- c(X).
b(b).

c(z).
```

This program becomes:

```
a(X) :- o_kbuild(1),b(X), c(X).
a(X) :- o_kbuild(2),d(X).
a(z) :- o_kbuild(3).

b(X) :- o_kbuild(1),c(X).
b(b) :- o_kbuild(2).

c(z) :- o_kbuild(1).
```

The primitive `o_kbuild` is implemented as a 'C' macro and treated as a goal by a stub Prolog procedure:

```
o_kbuild(N) :- pragma_c(o_kbuild1).
o_kbuild(_) :- pragma_c(o_kbuild2).
```

The 'C' macros defining `o_kbuild1` and `o_kbuild2` are as follows:

```
#define MAXORC  10000     /* maximum length of oracle            */

static int defer = 0;     /* flag to defer interrupt handling    */
static int skip = 0;      /* flag to skip prev port on interrupt */

static int orc[MAXORC];   /* array to hold orc as it is built    */

static int b_depth;       /* current BUILD or-depth              */

static int orc_l0;        /* depth limit L0                      */
static int orc_l;         /* depth limit L                       */
static int orc_g;         /* group count G                       */
static int orc_n;         /* unique processor number N           */
static int orc_s;         /* count of ports S                    */
static int orc_length;    /* length of oracle to follow          */

#define o_kbuild1
    { int index;
      ++b_depth;
      Deref(A(0),word,tag,adr)
      index = UnTag_INT(word);  /* index := argument to o_kbuild */
      if (b_depth <= orc_l0) { if (index != orc[b_depth]) fail;}
      else { if (b_depth <= orc_length)
               { if (index < orc[b_depth]) fail;
                 if (index > orc[b_depth]) orc_length = 0;
               }
             orc[b_depth] = index;
             if (b_depth == orc_l) /* if at partitioning depth...*/
               { if (skip) { skip = 0; fail; }
                 orc_s++;
                 if (orc_s % orc_g != orc_n) fail;
                 if (defer) { defer = 0; send_oracle(); fail; }
               }
           }
    }
```

```
#define o_kbuild2     /* called on backtracking through o_kbuild */
    {
      b_depth--;
      fail;
    }
```

Similar parallelisation primitives which exploit more complex transformations of the user program are possible (for example see Chapter 3). Also the primitive described above could equally be implemented wholly as a 'C' macro without the use of the stub Prolog procedure. The most efficient implementation might use a modified abstract machine. However, the implementation described above is portable to any Prolog compiler supporting embedded 'C' code, and the simple definition seems to produce an acceptable overhead of approximately 10%.

Bibliography

[1] Aït-Kaci, H. *The WAM: A (real) tutorial.* Tech. rep., Digital Paris Research Laboratory, Jan. 1990.

[2] Aït-Kaci, H., Lincoln, P., and Nasr, R. *Le Fun: Logic, equations and functions.* In *Proc. IEEE Intl. Symposium on Logic Programming, San Francisco*, pp. 17–23. 1987.

[3] Ali, K. *OR-parallel execution of Prolog on BC-machine.* In *Proceedings of the Fifth intl. conf. and symposium Logic Programming*, eds. R. A. Kowalski and K. A. Bowen, pp. 1531–1545. MIT Press, 1988.

[4] Ali, K., Karlsson, R., and Mudambi, S. *Performance of Muse on the BBN butterfly TC2000.* In *Parallel Execution of Logic Programs*, eds. A. Beaumont and G. Gupta, pp. 104–119. Springer-Verlag, 1991.

[5] 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.

[6] Alshawi, H. and Moran, D. B. *The Delphi model and some preliminary experiments.* In *Proceedings of the Fifth intl. conf. and symposium Logic Programming*, eds. R. A. Kowalski and K. A. Bowen, pp. 1578–1589. MIT Press, 1988.

[7] Appel, A. *Compiling with Continuations.* Cambridge University Press, 1992.

[8] Apt, K. R., de Bakker, J. W., and Rutten, J. J. M. M., eds. *Logic Programming Languages: Constraints, Functions, and Objects.* MIT Press, 1993.

[9] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, 1984.

[10] Barham, P. R. *Distributed DelPhi parallel Prolog.* Computer Science Tripos Part II Project, Computer Laboratory, University of Cambridge.

[11] BEAUMONT, A. *Scheduling strategies and speculative work.* In *Parallel Execution of Logic Programs*, eds. A. Beaumont and G. Gupta, pp. 120–131. Springer-Verlag, 1991.

[12] BELLIA, M. AND LEVI, G. *The relationship between logic and functional languages: a survey.* Journal of Logic Programming, 3, pp. 217–236, 1986.

[13] BONNIER, S. *A formal basis for Horn Clause logic with external polymorphic functions.* Tech. Rep. 276, Dept. of Computer Science, Linköping University, Sweden, 1992.

[14] BONNIER, S. AND MALUSZYŃSK, J. *Towards a clean amalgamation of logic programs with external procedures.* In *Proceedings of the 5th Intl. Conf. and Symposium on Logic Programming*, eds. R. Kowalski and K. A. Bowen, vol. 1, pp. 311–326. MIT Press, 1988. Re. S-Unification.

[15] BOSCO, P. G., GIOVANNETTI, E., AND MOISO, C. *Narrowing vs. SLD-Resolution.* Theoretical Computer Science, 59, pp. 3–23, 1988.

[16] BOWEN, D. L., BYRD, L., PEREIRA, F. C. N., PEREIRA, L. M., AND WARREN, D. H. D. *SICStus Prolog User's manual.* Swedish Institute of Computer Science, Apr. 1994.

[17] BRIAT, J., FAVRE, M., GEYER, C., AND DE KERGOMMEAUX, J. C. *OPERA: OR-parallel Prolog system on supernode.* In Kacsuk and Wise [48].

[18] BUTLER, R., DISZ, T., LUSK, E., OLSON, R., OVERBEEK, R., AND STEVENS, R. *Scheduling OR-parallelism: an Argonne perspective.* In *Proceedings of the Fifth intl. conf. and symposium Logic Programming*, eds. R. A. Kowalski and K. A. Bowen, pp. 1590–1605. MIT Press, 1988.

[19] CECCHI, C., SARTINI, D., AND AIELLO, L. *Evaluating logic programs via set-valued functions.* In *Proceedings of the 4th Intl. Conf. on Logic Programming*, ed. J. Lassez, vol. 1, pp. 428–455. MIT Press, 1987.

[20] CHAKRAVARTY, M. M. T. AND LOCK, H. C. R. *The implementation of lazy narrowing.* In *Proc. 3rd Intl. Symposium Programming Language Implementation and Logic Programming*, pp. 123–134. Springer-Verlag, 1991.

[21] CHANG, C. AND LEE, R. *Symbolic logic and mechanical theorem proving.* Academic Press, 1973.

[22] CHEONG, P. H. AND FRIBOURG, L. *Implementation of narrowing: The Prolog-based approach.* In Apt *et al.* [8].

[23] Clark, K. and Gregory, S. *A relational language for parallel programming.* Tech. rep., Dept. of Computer Science, Imperial College, London, 1981.

[24] Clark, K. and Gregory, S. *PARLOG: Parallel programming in logic.* Tech. Rep. 84/4, Dept. of Computer Science, Imperial College, London, 1984.

[25] Clocksin, W. F. *Principles of the DelPhi parallel inference machine.* Computer Journal, 30(5), pp. 386–392, 1987.

[26] Clocksin, W. F. *The DelPhi multiprocessor inference machine.* In *Proc. 4th U.K. Conf. on Logic Programming*, ed. K. Broda, pp. 189–198. Springer-Verlag, 1992.

[27] Clocksin, W. F. *Clause and Effect, Prolog for the Working Programmer.* Springer-Verlag, 1997.

[28] Clocksin, W. F. and Alshawi, H. *A method of efficiently executing Horn Clause using multiple processors.* Tech. Rep. CCSC-3, SRI International (Cambridge Computer Science Centre), 1987.

[29] Clocksin, W. F. and Mellish, C. S. *Programming in Prolog, 3rd Edition.* Springer-Verlag, 1987.

[30] Codognet, P. and Diaz, D. *wamcc: Compiling Prolog to C.* Tech. rep., INRIA-Rocquencourt, France, 1995.

[31] Conery, J. and Kibler, D. *Parallel interpretation of logic programs.* Proc. ACM Conference on Functional Programming Languages and Computer Architecture, pp. 163–170, 1981.

[32] Cuhna, J., Medeiros, P., Carvalhosa, M., and Pereira, L. *Delta Prolog: a distributed logic programming language and its implementation on distributed memory multiprocessors.* In Kacsuk and Wise [48].

[33] Curry, H. B. *Grundlagen der kombinatorischen Logik.* American Journal of Mathematics, 52, pp. 509–536, 789–834, 1930.

[34] Delgado-Rannauro, S. A. *OR-parallel logic computation models.* In Kacsuk and Wise [48].

[35] Deransart, P., Ed-Dbali, A., and Cervoni, L. *Prolog: The Standard.* Springer, 1996.

[36] Dincbas, M. and van Hentenryk, P. *Extended unification algorithms for the integration of functional programming into logic programming.* Journal of Logic Programming, 4, pp. 199–227, 1987.

[37] Disz, T., Lusk, E., and Overbeek, R. *Experiments with OR-parallel logic programs.* In *Proceedings of the 4th Intl. Conf. on Logic Programming*, ed. J. Lassez, vol. 2, pp. 576–600. MIT Press, 1987.

[38] Fribourg, L. *SLOG: A logic programming language interpreter based on clausal superposition and rewriting.* In *Proc. IEEE Intl. Symposium on Logic Programming*, pp. 172–184. IEEE Computer Soc. Press, 1985.

[39] Genesereth, M. R. and Nilsson, N. J. *Logical Foundations of Artifical Intelligence.* Morgan Kaufman, 1986.

[40] Gupta, G. and Santos Costa, V. *Cuts and side-effects in and-or parallel Prolog.* Tech. rep., Lab. for Logic and Databases, Dept. of Computer Science, New Mexico State University, USA, 1992.

[41] Hanus, M. *The integration of functions into logic programming: from theory to practice.* Journal of Logic Programming, 19,20, pp. 583–628, 1994.

[42] Hanus, M., Antoy, S., Kucken, H., and López-Fraguas, F. *Curry, an integrated functional logic language.* Tech. rep., RWTH Aachen, Germany, 1997.

[43] Henderon, F., Conway, T., Somogyi, Z., and Ross, P. *The Mercury Language Reference Manual.* University of Melbourne, 1995.

[44] Hindley, J. R. and Seldin, J. P. *Introduction to Combinators and λ-Calculus.* Cambridge University Press, 1986.

[45] Hoare, C. A. R. *Communicating Sequential Processes.* Prentice Hall, 1985.

[46] Hullot, J. M. *Canonical forms and unification.* In *Proc. of the Fifth Conference on Automated Deduction, Les Arcs, France*, vol. 87, pp. 318–334. Springer-Verlag, Jul. 1980.

[47] Josephson, A. and Dershowitz, N. *An implementation of narrowing.* Journal of Logic Programming, pp. 57–77, 1989.

[48] Kacsuk, P. and Wise, M., eds. *Implementations of Distributed Prolog.* Wiley, 1992.

[49] Klein, C. S. *Exploiting OR-Parallelism in Prolog using Multiple Sequential Machines.* Ph.D. thesis, Computer Laboratory, Cambridge University, England, February 1991. Reprinted as Technical Report No. 216.

[50] Lloyd, J. *Foundations of Logic Programming.* Springer-Verlag, 1984.

[51] Lloyd, J. W. *Combining functional and logic programming languages.* In *Proc. 1994 Intl. Logic Programming Symposium.* 1994.

[52] Lusk, E., Warren, D. H. D., Haridi, S., *et al. The Aurora OR-parallel Prolog system.* New Generation Computing, 7, pp. 243–271, 1990.

[53] Maluszyński, J., Bonnier, S., Boye, J., Kluźniak, F., Kågedal, A., and Nilsson, U. *Logic programs with external procedures.* In Apt *et al.* [8]. Re. S-Unification.

[54] Masuzawa, H., Kumon, K., Itashiki, A., Satoh, K., and Sohma, Y. *Kabu Wake parallel inference mechanism and its evaluation.* Proc. Fifth Generation Computer Conference, pp. 955–962, 1986.

[55] Milner, R., Tofte, M., and Harper, R. *The Definition of Standard ML.* MIT Press, 1990.

[56] Moreno-Navarro, J. J. and Rodriguez-Artalejo, M. *Logic programming with functions and predicates: The language Babel.* Journal of Logic Programming, 12, pp. 191–223, 1992.

[57] Naish, L. *Adding equations to NU-Prolog.* In *Proceedings of the 3rd Intl. Symposium Programming Language Implementation and Logic Programming*, eds. J. Maluszyński and M. Wirsing, pp. 15–26. Springer-Verlag, Aug. 1991.

[58] Naish, L. *Higher-order logic programming in Prolog.* Tech. Rep. 96/2, Dept. of Computer Science, University of Melbourne, Australia, 1996.

[59] Newmarch, J. D. *Logic Programming: Prolog and Stream Parallel Languages.* Prentice Hall, 1990.

[60] Paulson, L. C. *ML Exercise Sheets, Part 1A CST and Mathematics with Computer Science.* Tech. rep., Computer Laboratory, Cambridge University, England, 1988.

[61] Paulson, L. C. *ML for the Working Programmer.* Cambridge University Press, 1991.

[62] Paulson, L. C. and Smith, A. W. *Logic programming, functional programming, and inductive definitions.* In *Extensions of Logic Programming*, ed. P. Schroeder-Heister, LNAI 475, pp. 283–310. Springer, 1991.

[63] Reddy, U. S. *Transformation of logic programs into functional programs.* In *Proc. IEEE Intl. Symposium on Logic Programming*, pp. 187–196. 1984.

[64] Reddy, U. S. *Narrowing as the operational semantics of functional languages.* In *Proc. IEEE Intl. Symposium on Logic Programming, Boston*, pp. 138–151. 1985.

[65] Réty, P., Kirchner, C., Kirchner, H., and Lescanne, P. *NARROWER: a new algorithm for unification and its application to Logic Programming.* In *Proc. 1st Conference on Rewriting Techniques and Applications*, vol. 202, pp. 141–157. Springer-Verlag, 1985.

[66] Saraswat, S. *Performance Evaluation of the Delphi Machine.* Ph.D. thesis, Computer Laboratory, Cambridge University, England, Dec. 1995. Reprinted as Technical Report No. 385.

[67] Schönfinkel, M. *Über die Bausteine der mathematischen Logik.* Mathematische Annalen, 92, pp. 305–316, 1924.

[68] Shapiro, E. *A subset of Concurrent Prolog and its interpreter.* In *Concurrent Prolog: Collected Papers*, ed. E. Shapiro. MIT Press, 1987.

[69] Somogyi, Z., Henderson, F. J., and Conway, T. C. *Mercury, an efficient purely declarative logic programming language.* Tech. rep., Dept. of Computer Science, University of Melbourne, Australia, 1995.

[70] Stickel, M. *A Prolog technology theorem prover: Implementation by an extended Prolog compiler.* J. Auto. Reas., 4(4), pp. 353–380, 1988.

[71] Tick, E. *Parallel Logic Programming.* MIT Press, 1991.

[72] Tinker, P. and Lindstrom, G. *A performance-oriented design for OR-parallel logic programming.* In *Proceedings of the 4th Intl. Conf. on Logic Programming*, ed. J. Lassez, vol. 2, pp. 601–615. MIT Press, 1987.

[73] Warren, D. H. D. *An abstract Prolog instruction set.* Tech. Rep. TN-309, SRI International, Menlo Park, CA, 1983.

[74] Warren, D. H. D. *The SRI model for OR-parallel execution of Prolog - abstract design and implementation issues.* In *Proc. Symposium on Logic Programming.* IEEE Computer Society Press, Los Alamitos, California, 1987.

[75] Wise, M. *Prolog Multiprocessors.* Prentice Hall International, 1986.

[76] Wrench, K. L. *A Distributed AND-OR Parallel Prolog Network.* Ph.D. thesis, Computer Laboratory, University of Cambridge, Dec. 1990. Available in summary form as Technical Report No. 212.