Technical Report

UCAM-CL-TR-266
ISSN 1476-2986

Number 266

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Automatic exploitation of OR-parallelism in Prolog

Carole Klein

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

AUTOMATIC EXPLOITATION OF OR-PARALLELISM IN PROLOG

Carole Klein

Computer Laboratory
University of Cambridge
New Museums Site
Pembroke Street
Cambridge
CB2 3QG
England

e-mail address:
csk@cl.cam.ac.uk

Abstract

A path through a search space can be defined by a sequence of integers called an oracle. The Delphi Machine consists of a network of individual workstations co-operating to execute a Prolog program. Using oracles, these machines automatically partition the search space between them, thereby exploiting OR-parallelism. This report provides a brief description of the tree-searching algorithms (control strategies) implemented in the Delphi machine.

((Running Title))

AUTOMATIC EXPLOITATION OF OR-PARALLELISM

Introduction

As an alternative to the use of special architectures such as shared memory or multiprocessor machines [1,2,3], this paper describes a parallel Prolog implementation which operates over a network of processors.

It describes research which was completed in 1990, and includes the results obtained on a small network (20 workstations).

In addition to the speed-ups shown in the results, the system benefits from robustness. It can run on any number of processors and continue to run even when machines are rebooted. During the test period, several workstations in the network were damaged by fire, but this research was able to continue by running the programs on the remaining machines.

Oracles

If the branches at each node of a tree are numbered, the location of any node within the tree can be identified by a unique sequence of numbers called an oracle [4].

At each node of the tree in Figure 1, the branches are numbered from left to right. The oracle {4,1,2,1} leads to the node indicated.

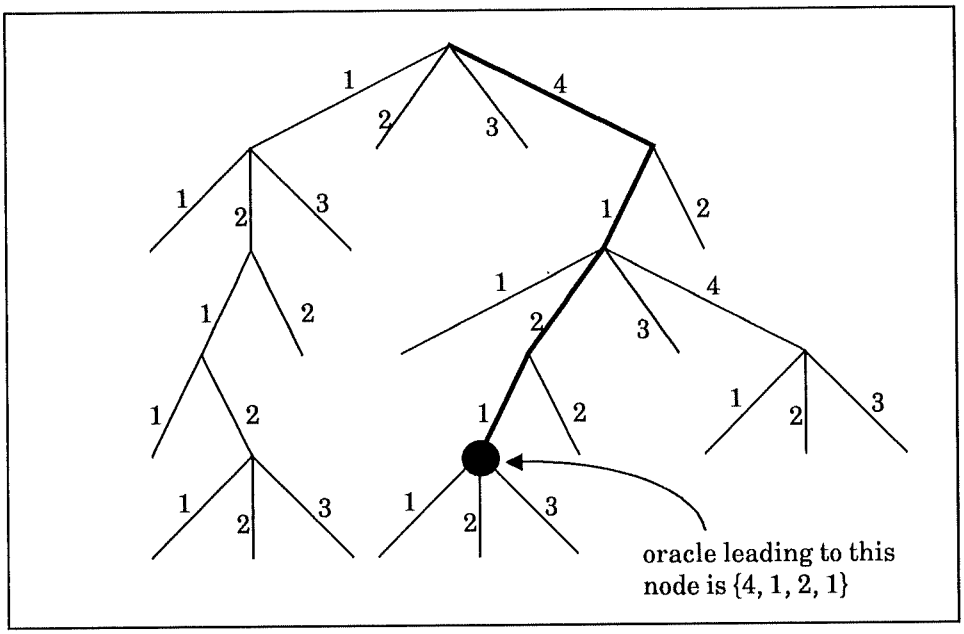


Figure 1 Each node within the tree can be identified by a unique sequence of numbers, an oracle

An oracle is therefore a path leading to a particular location in a search space. It can be followed by picking - at each node - the clause that corresponds to the next number in the oracle sequence.

As a processor follows an oracle, it creates the environment associated with each of the nodes along the path (because it is executing the program as it goes). So an oracle is also a concise means of specifying the environment at a particular node.

Prolog Search: the OR-only Tree

The execution of a Prolog program can be represented by an AND/OR tree. As an example, Figure 2 shows the AND/OR tree for the following program:

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

with goal clause `: - g(X,Y).` [5]

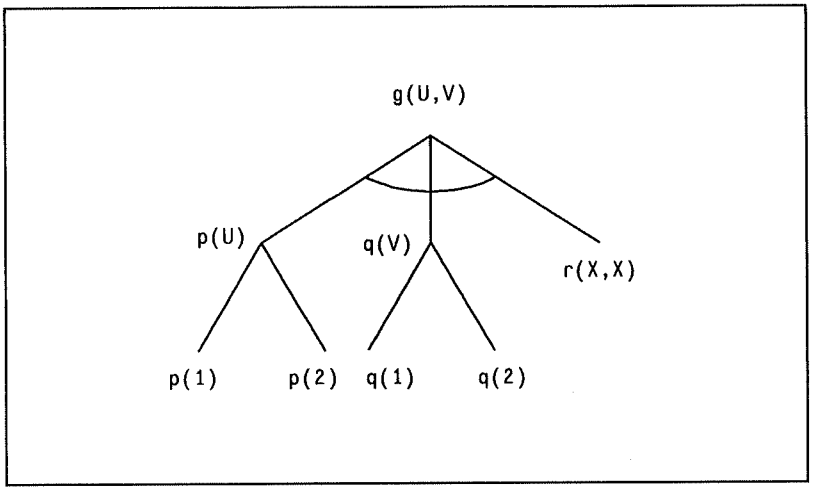


Figure 2 An example AND/OR Tree

Conceptually, a Prolog search may be thought of as a depth-first, left-to-right search of the associated OR-only tree. Figure 3 shows the creation of this tree, graphically.

It should be stressed that the program itself is not duplicated in memory. The OR-only tree is just a graphical representation of the Prolog search mechanism.

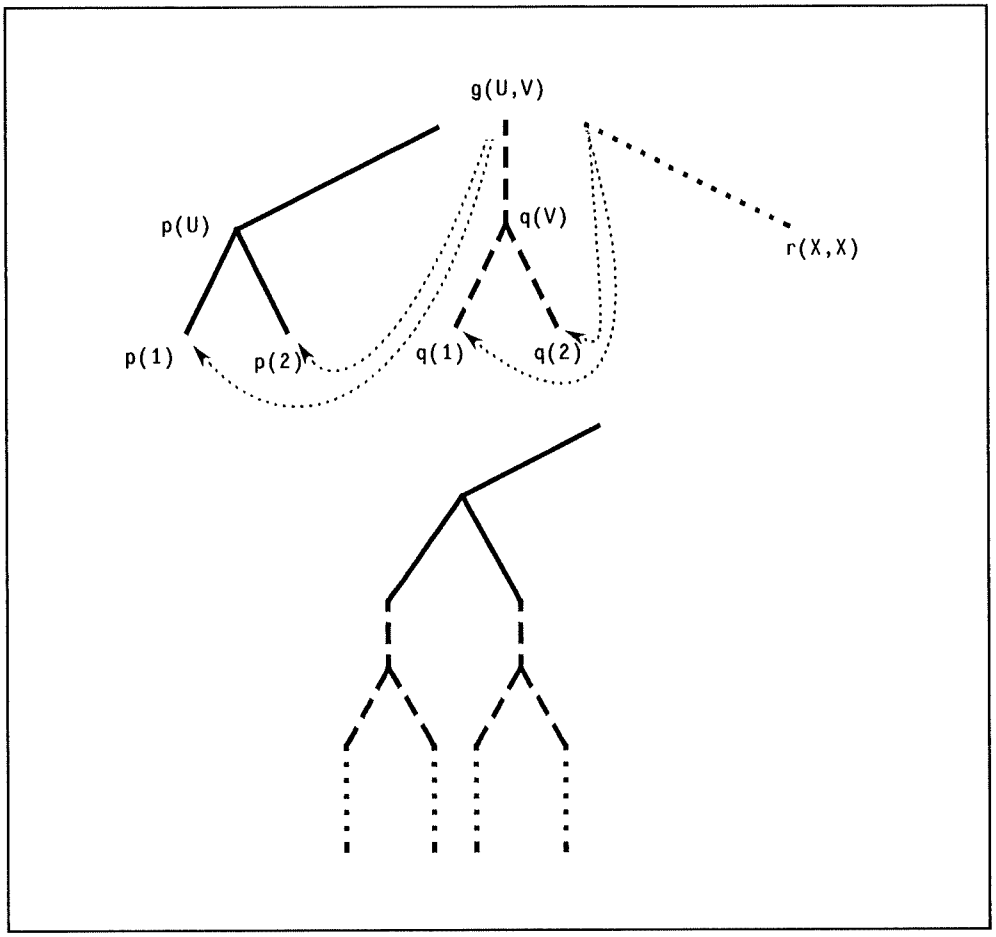


Figure 3 Creation of an OR-only Tree : the three AND branches are separated and copied

The OR-parallel search mechanisms described in this paper are illustrated using a condensed (but equivalent) tree representation. Since AND branches do not represent a choice, they are automatically executed and do not appear on the condensed tree. The tree created in Figure 3 thus becomes the tree in Figure 4.

All the nodes (choice points) in the tree are now potential points for exploiting OR-parallelism. Since the bindings at OR nodes are independent of one another, the branches of an OR-only tree can be partitioned amongst a number of processors and executed simultaneously.

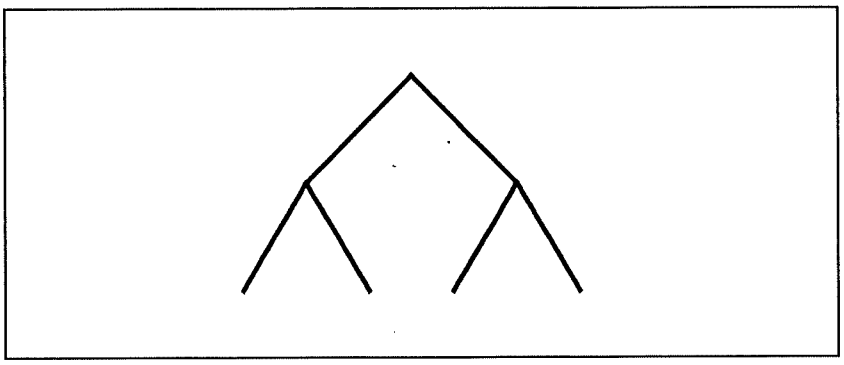


Figure 4 A concise OR Tree

The Delphi Machine

The Delphi Machine was first suggested by Clocksin and Alshawi [4]. Instead of relying on a special architecture, or using shared memory multiprocessors, they devised a system that could use the equipment then available at Cambridge University Computer Laboratory. This was a number of networked workstations operating independently to execute a single program.

The Delphi Machine consists of a control process (Controller) and any number of equivalent Prolog processes (Processor). Clocksin and Alshawi describe a theoretical Delphi: their Controller generates oracles according to certain control strategies. It allocates each Processor an oracle to follow. When a Processor reaches the end of its oracle, it reports back a result. It is then sent another oracle to follow. The cycle is repeated with all the Processors until the whole tree has been explored.

Applying this to the tree in Figure 4, four oracles describe the entire search space, namely {1,1} {1,2} {2,1} and {2,2}, and these can be assigned to separate Processors for execution. However, this method assumes that the Controller already knows the shape of the tree.

This paper describes a set of Delphi control strategies which do not rely on the generation of oracles by the Controller, but allow the individual Processors to generate them automatically. The results show that these strategies make the Delphi Machine a practical method for exploiting OR-parallelism.

Automatic Partitioning of the Tree

In the Automatic Partitioning strategy, each of several Processors takes a section of the tree for itself and searches this subtree as a serial Prolog process. At the same time, each keeps a record of its path through the tree. This path is a list of each of the clauses that lead from the root of the tree to the current node, in other words, an oracle.

At the start of a Delphi session, the source Prolog code is compiled and an identical copy of the compiled code is sent to each of the participating Processors (in general, each workstation runs a single Prolog process). In addition, each Processor is sent the initialisation parameters it needs to execute Automatic Partitioning, and once these are received, immediately starts executing the program.

Manipulation of two variables, G and N, achieves Automatic Partitioning. There is no need for further communication amongst the Processors. G is the group counter, a record of the total number of Processors in the group. In addition, each Processor is assigned an identifier number, N - the first Processor is assigned 1, the second 2, and so on, up to the last Processor in the group, which is assigned G.

The Processors begin at the root of the tree. As this is a choice point, they begin splitting up the search space. They know how many Processors are at the node (G),

they know how many branches there are to choose from (this is a feature of the Delphi compiler written for this implementation), and they know their own number within the group (N).

At any choice point there are four possibilities:

only one Processor available
fewer Processors than branches
equal numbers of Processors and branches
more Processors than branches

In the first case, only one Processor available, the Processor behaves in the same way as a serial Prolog process.

In the second case, fewer Processors than branches, the rightmost branch is taken by the highest numbered Processor ($N=G$), the next rightmost by the next highest ($N=G-1$), and so on, until only Processor 1 remains. It takes the rest of the tree to execute on its own. Each Processor's group counter (G) is now modified to reflect the new number of Processors proceeding along each branch.

If the number of Processors equals the number of branches, each Processor takes the clause associated with its number. Each group counter is set to 1 to reflect the fact that only one Processor is proceeding along each branch.

If there are more Processors than branches, each Processor takes the clause associated with its number, and the extra Processors take the rightmost branch. Their N and G variables are set accordingly.

The splitting is repeated in each group, at each node, until each Processor is executing its own unique portion of the search space.

Figure 5 shows an example of Automatic Partitioning between 6 Processors, and charts the changes to the values of G and N as execution proceeds.

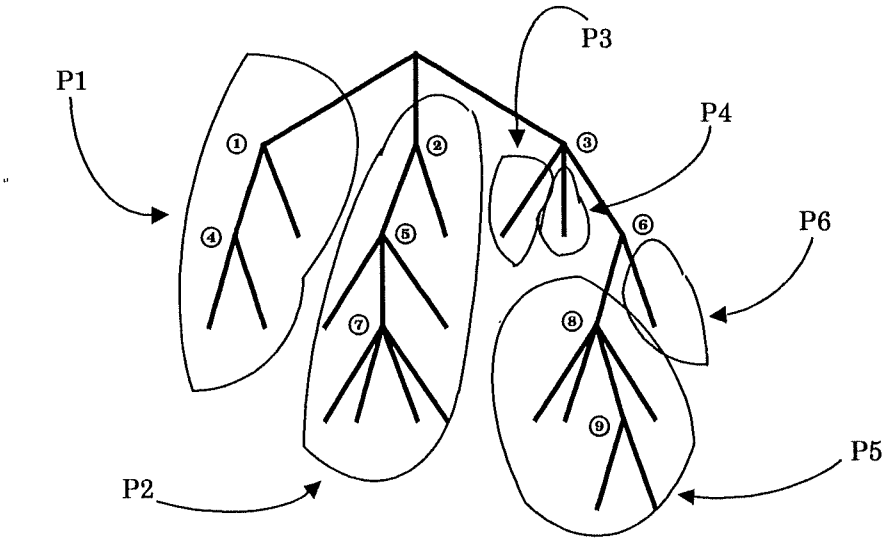
Figure 6 shows how this same tree would be divided amongst from one to six Processors.

This is a right-biased strategy - it assumes that the rightmost branches of the tree will always be the longest, and assigns as many Processors as possible to the right branch. Other possibilities are a left-biased strategy or one which divides the Processors more equally between the branches.

The Reassign Jobs Strategy

Whatever algorithm is used to partition the search space, some sections of the tree will prove less branchy than others, so some Processors will finish their search before the others and be left idle. The Controller maintains all idle Processors in an idle process queue. The reassign jobs strategy allows the Controller to send idle Processors to the portions of the tree which still remain to be searched. This is accomplished with the minimum of communication.

When a Processor is uniquely searching some part of the tree, its G and N values are both equal to one. Under these conditions, when it reaches a branching point, the Processor can check-in to the Controller to request help with its workload. As mentioned earlier, as a Processor executes the tree, it keeps a record of the path to its current position in the form of an oracle. A check-in consists of sending this oracle to the Controller. The Controller responds by sending back an integer, telling the Processor how many idle Processors are waiting in the queue. If this number is zero, the Processor continues searching the tree on its own. If the



Node	P1		P2		P3		P4		P5		P6	
	N	*G*	*N*	*G*	*N*	*G*	*N*	*G*	*N*	*G*	*N*	*G*
initialisation	1	6	2	6	3	6	4	6	5	6	6	6
root	1	1	1	1	1	4	2	4	3	4	4	4
①	1	1										
②			1	1								
③					1	1	1	1	1	2	2	2
④	1	1										
⑤			1	1								
⑥									1	1	1	1
⑦			1	1								
⑧									1	1		
⑨									1	1		

Figure 5 An example of Automatic Partitioning

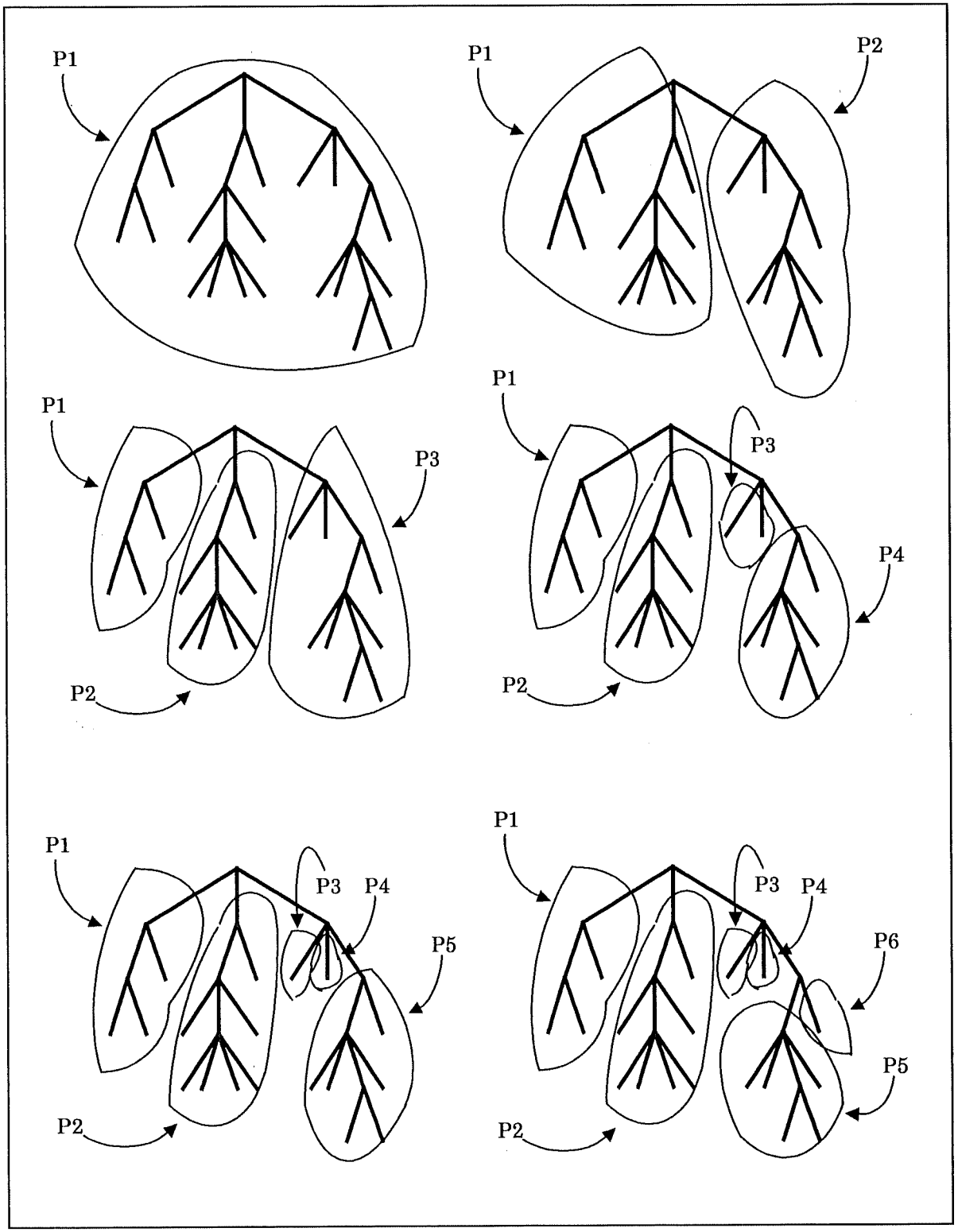


Figure 6 Automatic Partitioning between from 1 to 6 Processors

number is greater than zero, then the Processor becomes part of a group of Processors that will automatically divide the work from that point.

Reassign jobs is an extension of automatic partitioning. Instead of starting at the root of the tree, the group of Processors is given an internal node as a starting point. This involves giving each idle Processor :

> a new group count (G) equal to the number of idle Processors plus one (the Processor that checked in)
>
> a new identifier number (N), and
>
> the oracle sent by the Processor that checked in, which describes the path to the new starting point.

Figure 7 shows this graphically. Assume that a Processor has reached the node indicated and has checked in by sending oracle {2,1,2,1}. Assume that there 5 Processors on the idle queue. The Controller sends the value 5 to the checking-in Processor which automatically sets N equal to 1 and begins executing the leftmost branch. The Controller sends each of the idle Processors three pieces of information: the group count 6, its new identifier number 2-6, and the oracle {2,1,2,1}. The idle Processors follow this oracle, reach the new starting point directly (without any backtracking), and split the rest of the tree between them.

Notice that because the tree beneath the new starting point happens to be identical to the tree in Figure 5, it is automatically partitioned in exactly the same way.

This cycle of searching, checking-in, becoming idle, and being reassigned continues until the entire search space has been covered. The exact frequency of

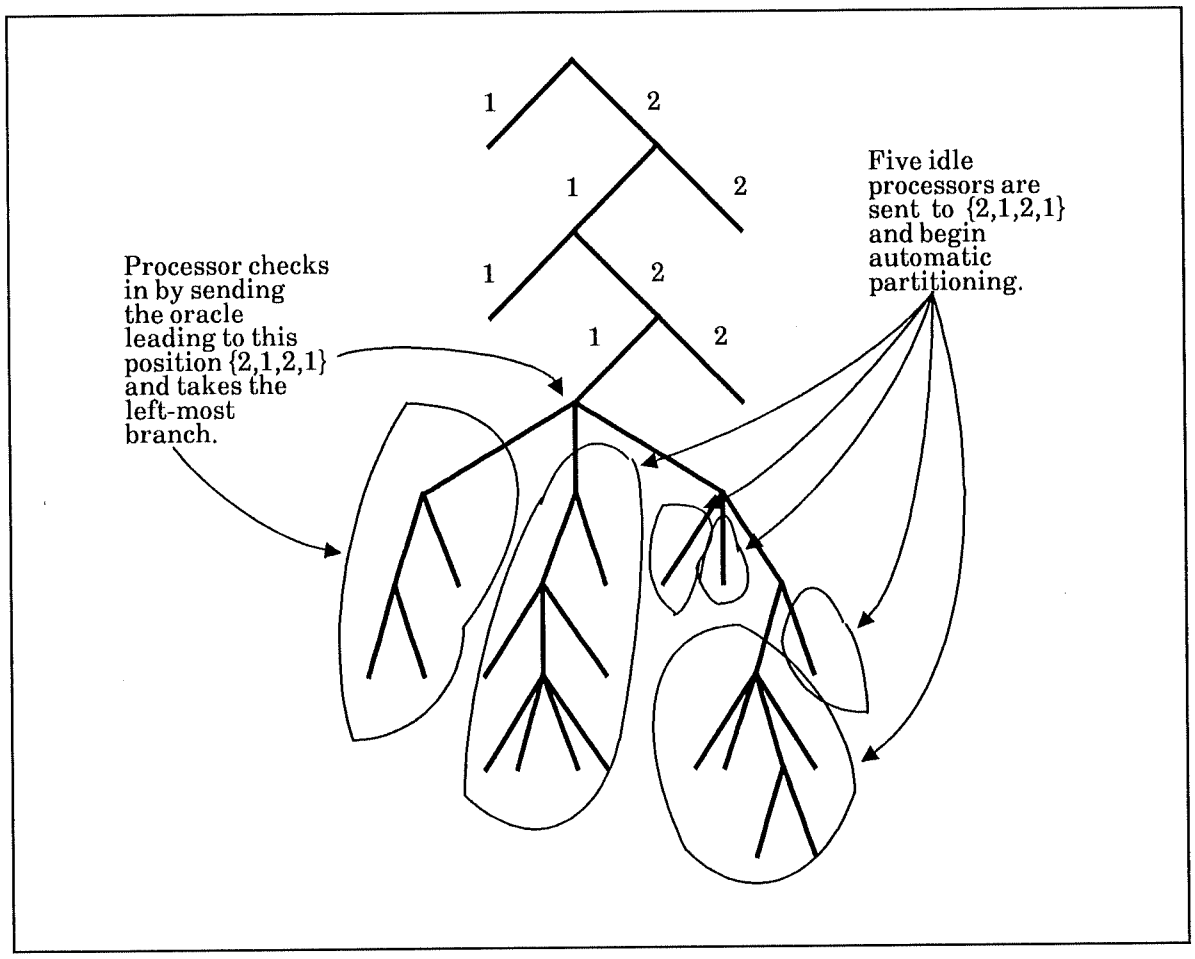


Figure 7 The Reassign Jobs strategy

check-ins can be regulated by setting a check-in interval. If this is set low, the frequency of check-ins will be high, and vice versa.

Results

The first program run on the Delphi Machine was the 8-Queens Problem. The graph in Figure 8 shows the execution times (in seconds) for a network of from one to 20 Processors.

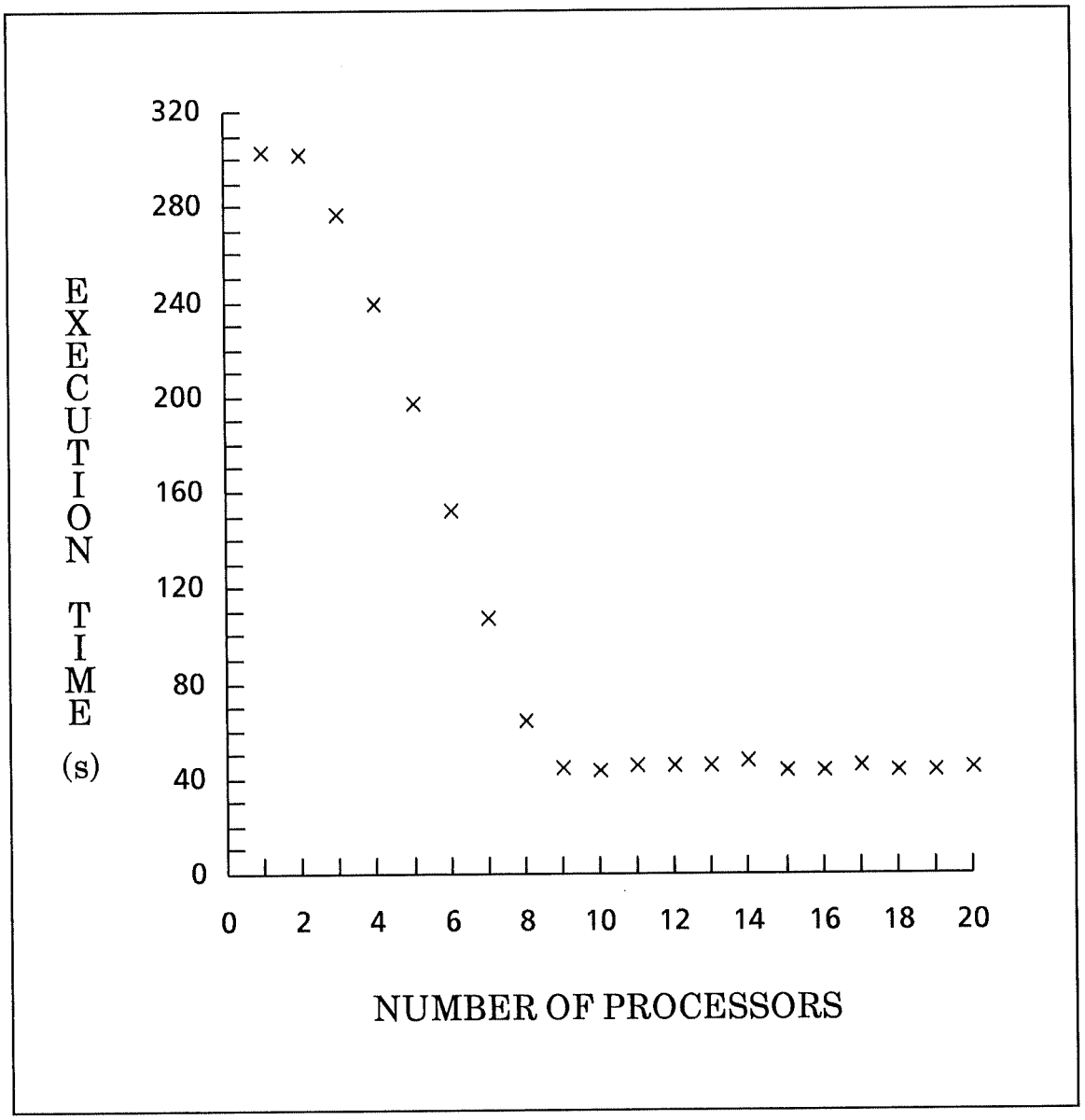


Figure 8 The 8-Queens Problem running on from 1 to 20 Processors

The relative speed-up for a number of different problems is shown in Table 9. The first two columns show the results for C-Prolog (Version 1.4 [6]) and Cosmic Prolog. Cosmic Prolog is a modified version of SB-Prolog Version 2.2, a public domain Prolog developed at the State University of New York at Stony Brook. (Cosmic Prolog was used to develop the Delphi Prolog System). The third column shows the results for a 20 Processor Delphi Machine running on a network

(Ethernet) of μVAXes. The fourth column shows execution time for a single Processor Delphi, and the final column shows the relative speed-up between the two.

Problem	C-Prolog	Cosmic Prolog	Delphi (20 Processors)	Delphi (1 Processor)	relative speed-up
parser-2	38	38	11	58	5.27
parser-3	108	100	25	177	7.08
parser-4	390	354	73	461	6.32
adder	1192	805	154	1814	11.78
pentominoes	19005	13474	2521	26798	10.63
8-queens	411	175	42	301	7.17
9-queens	1994	841	127	1405	11.06
10-queens	10245	4560	450	8137	18.08

Table 9 Comparative results

The timings include all communications, from the initialisation phase to all solutions being collected in a global log file, and have been rounded to the nearest second.

Because of overheads imposed by the Delphi algorithm a program executed on a single Processor Delphi Machine runs at approximately half the speed of the same

program running on an unmodified Prolog system. However, as more Processors are added, the speed-up obtained is nearly linear for programs that include large amounts of OR-parallelism.

A complete description of the problems run on Delphi, together with source code, and full results of numerous control strategies can be found in Klein [7].

References

[1]
Ciepielewski, A., Haridi, S., and Hausman, B., OR-parallel Prolog on shared memory multiprocessors, *The Journal of Logic Programming*, **7**(2):125-147 (1989).

[2]
Lusk, E., Butler, R., *et alia*, The Aurora Or-parallel Prolog System, *New Generation Computing*, 7:243-271 (1990).

[3]
Harsat, A. and Ginosar, R., CARMEL-2: A Second Generation VLSI Architecture for Flat Concurrent Prolog, *New Generation Computing*, 7:197-218 (1990).

[4]
Clocksin, W.F. and Alshawi, H., A Method for efficiently executing horn clause programs using multiple processors, *New Generation Computing*, 5:361-376 (1988).

[5]
Clocksin, W.F., Principles of the DelPhi parallel inference machine, *Computer Journal*, **30**(5):386-392 (1987).

[6]
Pereira, F., C-Prolog User's Manual 1.4, SRI International, Menlo Park, California, 1984.

[7]
Klein, C.S., Exploiting OR-parallelism in Prolog using multiple sequential machines, Technical Report No.216, University of Cambridge Computer Laboratory, Cambridge, 1991.

Figure Legends

Figure 1: Each node within the tree can be identified by a unique sequence of numbers, an oracle.

Figure 2: An example AND/OR tree.

Figure 3: Creation of an OR-only tree: the three AND branches are separated and copied.

Figure 4: A concise OR-tree.

Figure 5: An example of Automatic Partitioning.

Figure 6: Automatic Partitioning between from 1 to 6 Processors.

Figure 7: The Reassign Jobs strategy.

Figure 8: The 8-Queens Problem running on from 1 to 20 Processors.

Table 9: Comparative Results.