Distributed logic programming with oracles and kappa

Ian J. Lewis
University of Cambridge
Cambridge, England, CB2 3QG

Abstract

The recursive backtracking performed during the execution of a Prolog program can be interpreted as the traversal of a proof tree. An *oracle* is a sequence of clause indexes representing a path within that tree. Combined with the depth-first left-to-right search of standard Prolog, an oracle provides an effective means of communicating and recreating the state of an interrupted processor, denoting a subtree for search, uniquely identifying a returned solution, and dividing the search tree into two parts. This paper shows how those properties can be exploited in a simple yet effective OR-parallel Prolog system for distributed networks of general purpose workstations.

1 Introduction

Given a goal clause and a program, a Prolog system will systematically search the implied AND-OR proof tree. OR-parallel systems evaluate alternative choices in parallel, generally requiring the accumulated state to be communicated or shared among the processors working within a common subtree [1, 3, 6, 7, 9, 12]. The use of oracles, proposed by Clocksin in [5], provides a model for OR-parallel execution of logic programs in which no computation state (meaning constructed data structures and variable bindings) need be copied or referenced between processors. An *oracle* is a list of clause indexes defining a path in the transformed OR-only search tree of the program. The principle is illustrated in the sample program given in Figure 1, producing the AND-OR tree represented on the right. The left-to-right ordering of the conjunctive goals in the body of the first clause leads to the transformation of the AND-OR tree into the OR-only tree of Figure 2, in which the subtree representing each conjunctive subgoal is replicated under each terminal node of the left-most OR-only subtree. The execution of the Prolog program equates to the depth-first left-to-right traversal of this OR-only tree. In this paper, *search tree* will be taken to mean the idirected acyclic graph represented by this OR-only tree.
Each clause of the program can be assigned an index local to its procedure,

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

p(1).

p(2).

q(1).

q(2).

r(X,X).

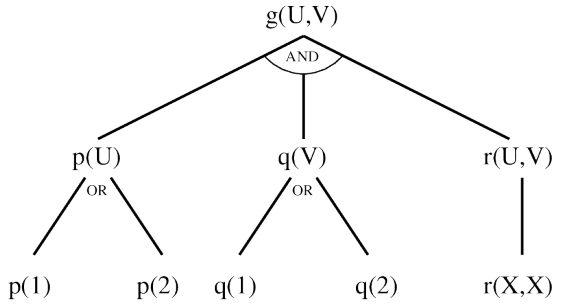

Figure 1: AND-OR search tree for goal clause g(U,V).

and each oracle is a list of these indexes representing a path from the root of the search tree. The principle of the DelPhi machine investigated by Clocksin and others [5, 10, 8] is that these paths can be articulated by one processor for execution by others, resulting in parallel search. Rather than the communication or sharing of data structures created during the execution of the program, *recomputation* is used to create the bindings valid for the assigned path. In practical situations, it is often more efficient to use computing cycles to recreate the environment at a given node in the search tree than to copy or share that environment between processors.

The OR-only tree in Figure 2 is annotated with the local clause indexes[1] and two oracles are illustrated. An *open oracle* defines a path leading to an intermediate node in the search tree, such as the oracle *[1,2]* leading to the node labelled B in the example. The oracle leading to the solution at the terminal node A is *[1,2,2,1]*. The processor receiving the oracle is called a *path processor*, and can be expected to execute and possibly extend the assigned oracle, communicating success or failure or returning a number of discovered open oracles. During execution, a path processor can maintain a list of clause indexes leading to its current point in the search tree, this list being its *current oracle*.

The DelPhi principle permits a wide range of distributed execution strategies, discussed in detail in [5, 10, 8]. With strategies based upon oracle generation and testing, each path processor can limit its search along the assigned path, reporting success, failure, or open. Other strategies can require the path processors also extend the assigned oracle by one or more indexes, or as with breadth-first partitioning described below, fully search

[1] Alternatively, the use of global clause indexes in an oracle, such as the entry-points of each clause, allows oracles to be followed efficiently but may be less suited to techniques involving oracle generation outside of the program. The use of bit strings for oracles, suggested by Clocksin in [5, 6], is most suited to these approaches.

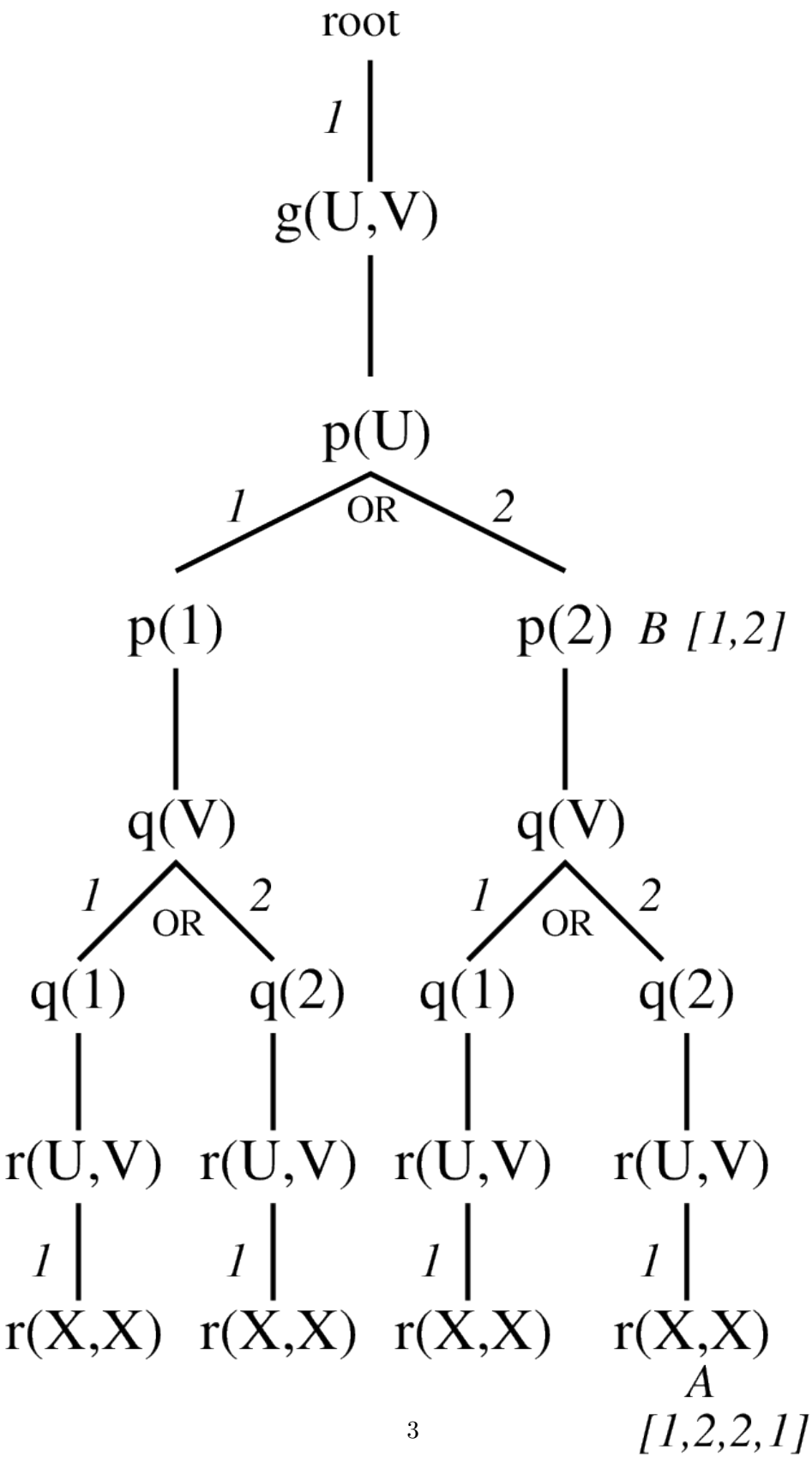

Figure 2: OR-only tree annotated with oracle indexes.

the subtree referenced by the oracle.
Thus oracles can be given a number of interpretations:

- An oracle defines a path in the OR-only search tree that can be executed by a processor, returning success, failure, or open. This interpretation is useful for parallelisation strategies using a control processor to generate oracles for execution by dependent path processors.

- An oracle uniquely defines a node within the search tree. For a terminal node indicating success, the oracle uniquely identifies the solution. The communication of work using oracles provides straightforward support for efficient recovery in the event of the loss of a path processor, and the labelling of solutions with their associated oracle permits the efficient reassignment of the affected work avoiding the possibility of duplicate solutions.

- An open oracle uniquely defines the root of subtree within the search tree, suggesting the strategy of breadth-first partitioning in which all the subtrees defined by open oracles of a fixed length are allocated to the available path processors for distributed search. This strategy is described in Section 2.

- When combined with the depth-first left-to-right search strategy of standard Prolog, an oracle divides the OR-only search tree into a left-part and a right-part. The current oracle of a busy path processor assigned a subtree for search divides that subtree into the parts already searched and yet to be searched.

The assignment of oracles to path processors for interpretation as references to unique subtrees for distributed search leads to the concept of *poisoned oracles* which can greatly reduce the efficiency of simple scheduling strategies. For example, a lengthy oracle leading to a very small subtree will cause a path processor to perform much recomputation culminating in relatively little useful work searching the allocated subtree. Equally, an oracle leading to a huge subtree in a strategy in which the path processor fully searches each assigned subtree without interruption will also limit the parallelisation efficiency.
The breadth-first partitioning (BFP) strategy described below provides effective parallelism for programs without greatly imbalanced search trees. The OR-parallel partitioning can be achieved with an extra-logical predicate of only 4 lines of 'C'. For problems with less evenly distributed work, the work splitting strategy described in Section 4 recursively applies BFP, avoiding the inefficiency caused by large poisoned oracles.

2 Breadth-First Partitioning

The breadth-first partitioning (BFP) strategy, first investigated by Saraswat in [10], is illustrated in Figure 3. The strategy uses the depth-first left-to-right search of standard Prolog, proceeding in two phases:

1. **Discovery of open oracles:** the search is limited to a specified depth limit L. All the open oracles found at this depth are accumulated into an oracle stack for allocation in the second phase. In Figure 3 the five discovered open oracles are labelled A...E.

2. **Selected subtree search:** for each oracle, a path processor follows the oracle and fully searches the referenced subtree. With the count of path processors in the group G, each path processor is allocated a unique processor N in the range $0 \ldots G-1$. The filter $i \bmod G = N$ is used to select the ith oracle for execution by path processor N. In the example of Figure 3, a system with two path processors numbered 0 and 1 will allocate oracles B and D to processor 0, and oracles A,C and E to processor 1.

The description given above suggests the generation of the oracles in a control processor, and the subsequent distribution of those oracles to the available path processors. Communication within the system can be reduced if the oracles are produced locally within each path processor, which then use the G and N parameters to select the subtrees for local search. A control processor is needed to start the processes with the correct parameters, collect the solutions, and report completion.
A number of benchmarks have been used to evaluate the performance of the BFP strategy, with detailed results in [10, 8]. A series of runs of the Pentominoes program [8] gives the speedup for $G = 1 \ldots 30$ in Figure 4. The single cpu runtime for the Pentominoes program is 446 seconds, with the maximum improvement reducing that time to approximately 37 seconds. The overhead of the oracle management with a simple implementation increases the single-cpu runtime by 9% compared to the underlying standard Prolog compiler.
The graph for the Pentominoes problem shows a general speedup, but reveals some issues not apparent with some more benign benchmarks, such as N-queens. For some group sizes, the speedup achieved for the sample problem was less than that achieved with fewer processors ($G = 12, 24$ and 30). This phenomenon is caused by the uneven distribution of the work in the subtrees beneath the open oracles found at the selected depth limit $L = 21$. At this depth limit, the Pentominoes problem has 848 open oracles, such that the largest group of 30 processors are assigned 28 or 29 open oracles each. The work allocated to the path processors benefits from averaging across the oracles assigned, but the allocation formula includes no estimate of the work

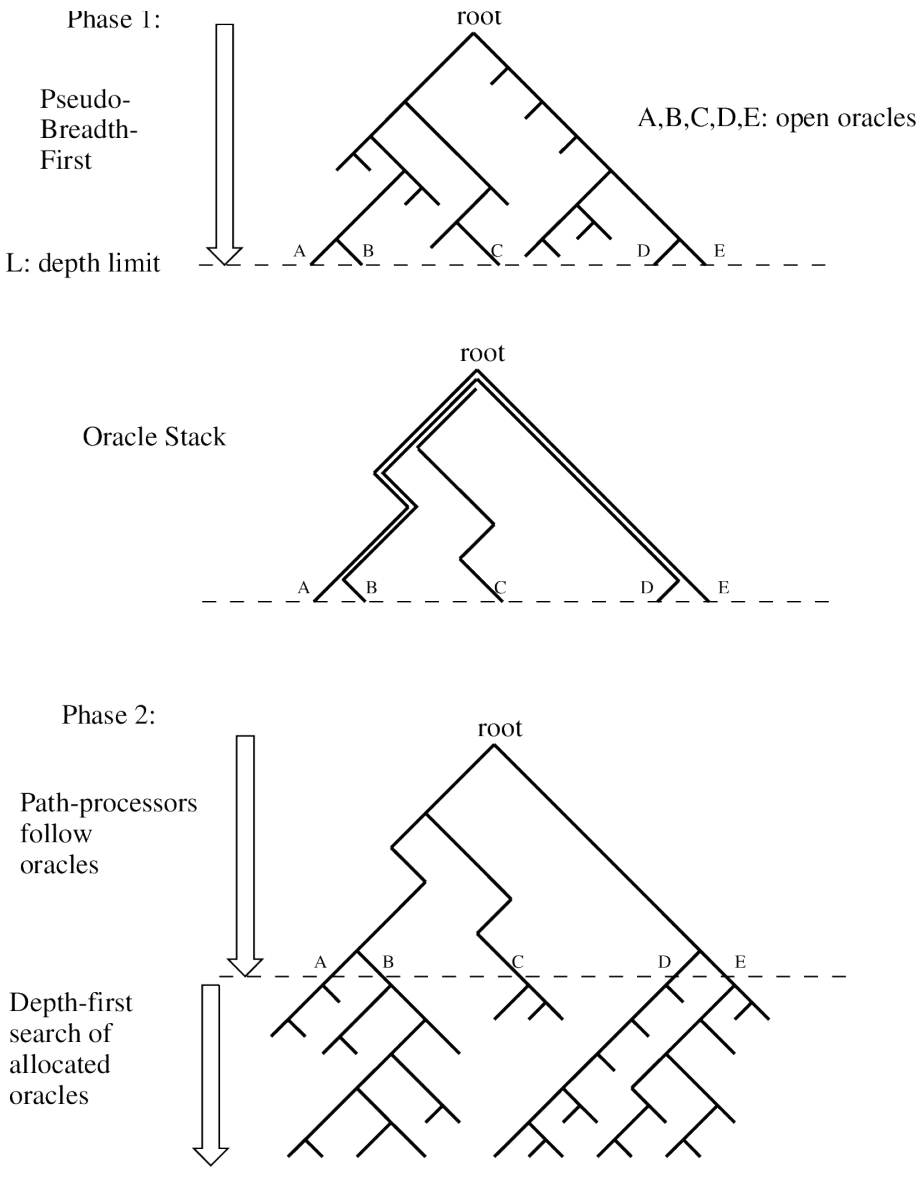

Figure 3: Oracles and Breadth-First Partitioning.

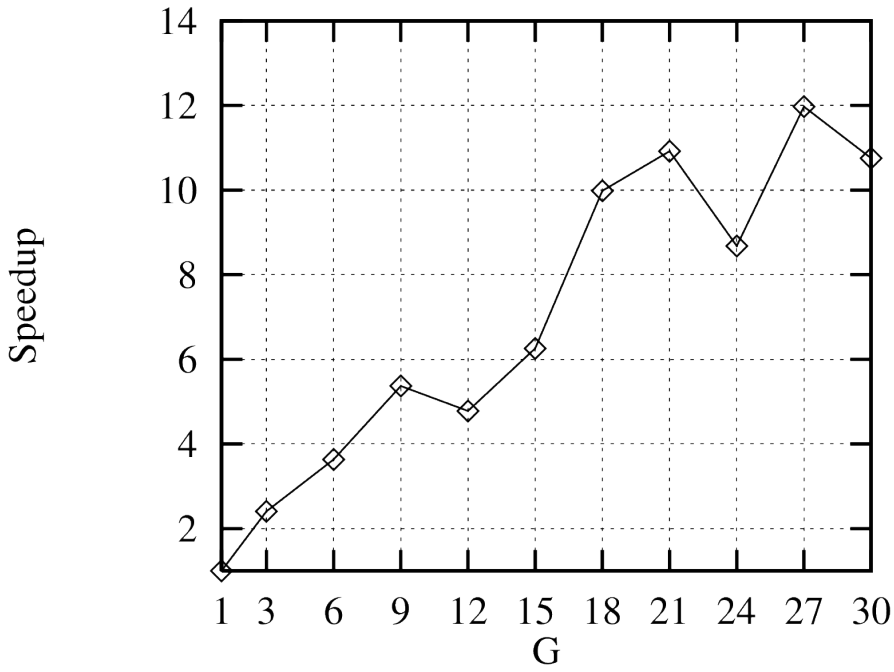

Figure 4: Speedup achieved for Pentominoes program.

in each subtree and some distributions may select several large subtrees for execution by a single path processor. In the Pentominoes problem and other benchmarks evaluated, typically fewer than 10% of the open oracles referred to over 90% of the work. The depth limit selected must assign enough oracles to each path processor so that all those with small subtrees can be quickly searched. The depth limited search of the first phase is effectively sequential, so the limit L must not be so large as to cause the first phase to dominate the overall runtime.

Figure 5 shows the speedup achieved for the Pentominoes program with the BFP strategy for a fixed processor group size $G = 30$, with a range of values for the partitioning depth limit L and the fixed modulo oracle allocation formula. For small values of L the speedup is limited by the small number of oracles discovered, such that some path processors receive none. At values of $L > 21$, the overhead of the first phase reduces the parallel speedup. The dashed lines in the graph of Figure 5 give the speedup performance for BFP implemented with a demand-based oracle allocation algorithm rather than the local fixed modulo formula. This technique benefits from a more balanced allocation of the work available at the selected depth limit, but the communication required for the transmission of each oracle for execution introduces a new overhead. The graph of Figure 5 shows the calculated speedups for demand-based oracle assignment, with communication latencies of 25ms and 250ms.

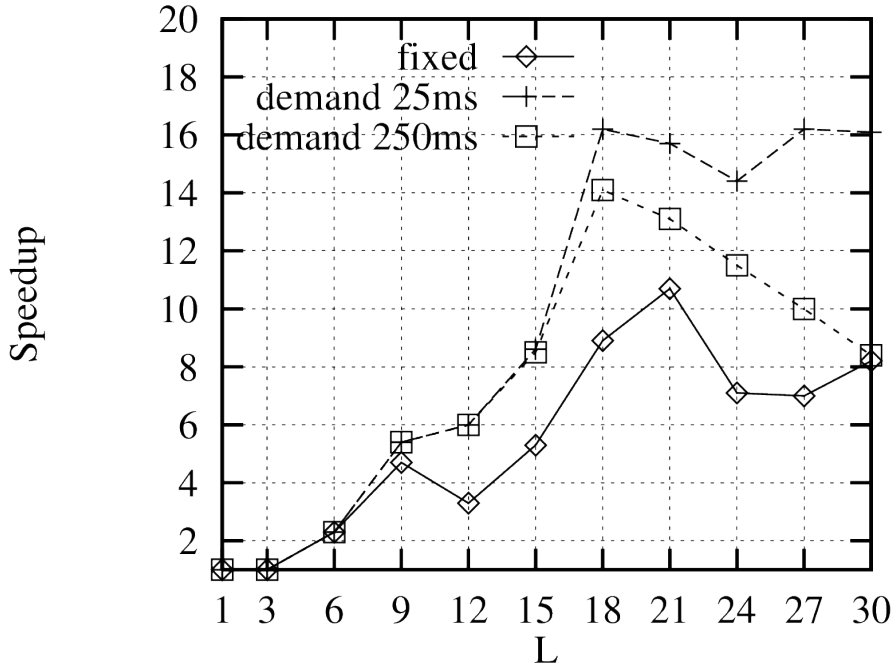

Figure 5: Effect of BFP depth limit on speedup.

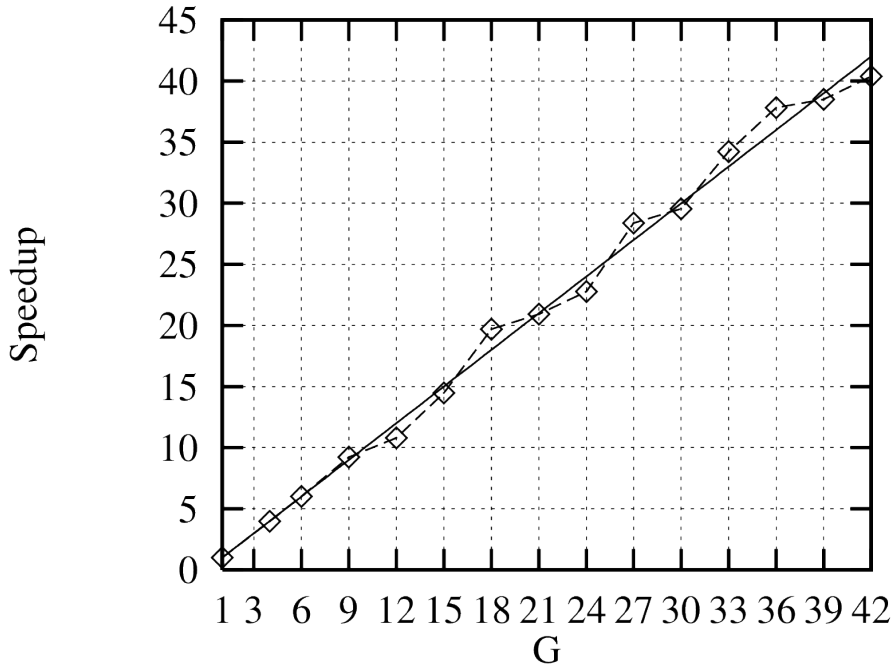

Figure 6: Speedup achieved for PTTP Overbeek example 4.

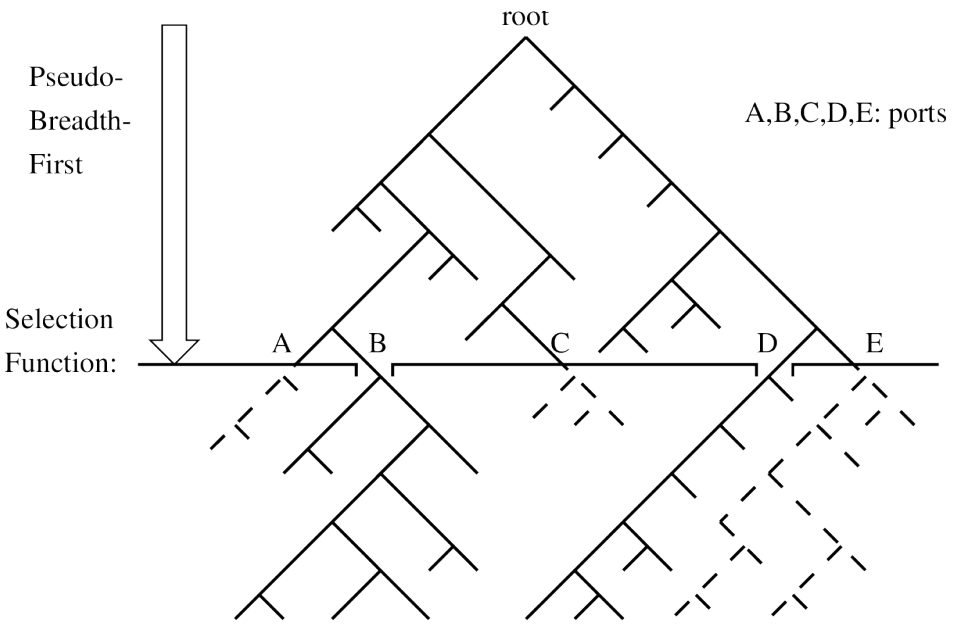

Figure 7: Breadth-first partitioning without oracles.

The breadth-first partitioning strategy described above is vulnerable to large imbalances in the work beneath each discovered open oracle, and requires the user selection of a partitioning depth limit. However, with these limitations some useful results are possible. Figure 6 gives the speedups achieved for $1 \ldots 42$ processors for the Prolog Technology Theorem Prover developed by Stickel given a problem from Overbeek [11]. The single-cpu runtime for the program was 6 hours and 6 minutes. The following section describes `kappa`, a four-line extra-logical predicate which was added to PTTP to produce the results shown.

3 Kappa: partitioning without oracles

The two phases of the breadth-first partitioning strategy given in Section 2 with the fixed modulo oracle assignment formula can be combined into one. Each path processor is given the program and the parameters G, N and L, and searches from the root to the depth limit L. Oracles are not accumulated, but a cumulative count is recorded of the number of times the depth limit is reached. This count is equal to the corresponding oracle number, and the modulo selection formula can be used to identify the subtrees which should be selected for search. The path processor searches each selected subtree as soon as its root is discovered during the depth limited search.
The process is illustrated in Figure 7, with the roots of the subtrees at the depth limit L labelled A,B,C,D and E as with the equivalent oracles for

BFP. No oracles are used for partitioning, and the nodes in the search tree at the depth limit L at which further search is possible are called *ports*. The resulting distributed search has much in common with BFP, but the recomputation of the search tree beneath the depth limit is reduced and no storage or communication resources are required for the oracles.
The partitioning can be achieved with the use of a simple extra-logical predicate `kappa`, the source for which is given in Section 6. If one-time partitioning is used as with BFP, the distributed program will have the same vulnerability to ports or oracles leading to huge subtrees, and a reasonable partitioning depth limit must be used.
The provision of one-time breadth-first partitioning through the introduction of the simple extra-logical proposition `kappa` allows rapid experimentation with OR-parallelism for existing Prolog programs. The proposition can be automatically added to every clause, or the programmer can annotate just one procedure.

4 Work splitting with oracles and `kappa`

The strategy of breadth-first partitioning using oracles, or the similar distributed behaviour produced through the use of `kappa`, can provide good speedup for programs with reasonably balanced search trees. The integration of the support for oracles into the partitioning predicate `kappa` broadens the range of programs for which the parallelising technique is effective.
A graphical representation of the recursive partitioning technique, called splitting with oracles and `kappa` (SOK), is given in Figure 8. Each path processor loads the user program on startup, and the assignment of work to each path processor is specified with:

- An oracle and a prefix length L. The first L indexes of the oracle lead to the root of a subtree within which the search is to be constrained. The root of the subtree is labelled B in Figure 8. The remaining indexes of the oracle from B to X, divide the subtree into two parts.
- An incremental depth limit L', at which breadth-first partitioning as described in Section 3 is to take place.
- A parameter G giving the number of path processors in the group allocated to the subtree at B.
- A unique processor number N.

Each path processor will *follow* the assigned oracle from the root of the search tree to B. The remainder of the oracle is treated as a left bound to the search within the subtree at B. With this left bound, breadth-first partitioning is used within the subtree, and the discovered ports at the

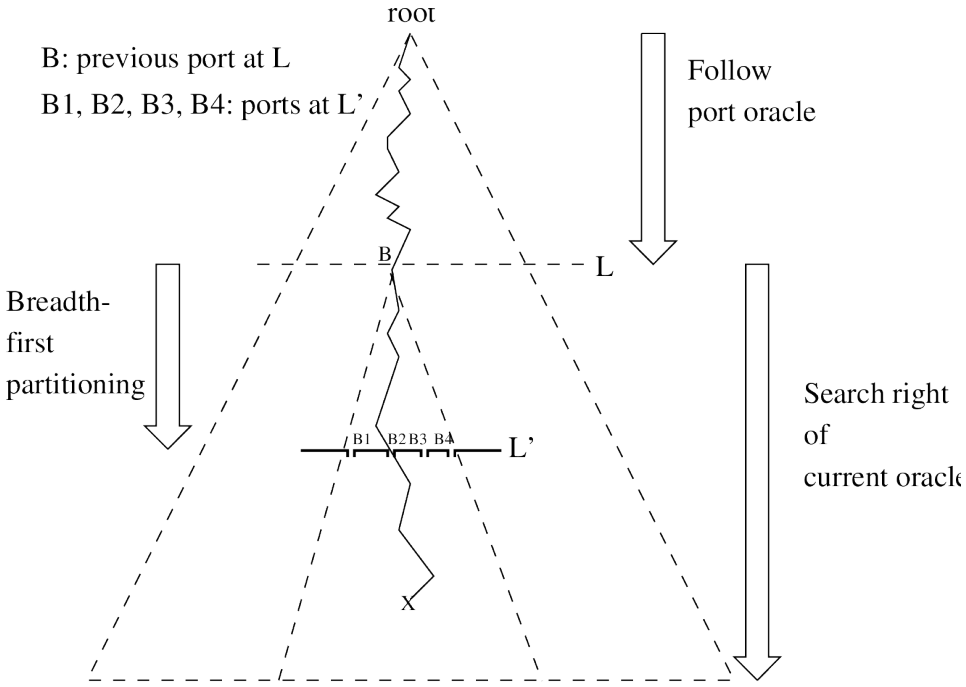

Figure 8: Work splitting.

incremental depth limit L', labelled B2, B3 and B4 in Figure 8, are allocated modulo G to the path processors in the new group. The path processor with unique processor number $N = 1$ will be allocated the first port discovered at B2, and will first search the subtree with its root at B2 with the oracle B2-X at a left bound to the search.
While executing the search, a busy path processor is liable to be interrupted, at which point it communicates its current oracle and current partitioning depth limit, and aborts the search of its current subtree. The interrupted path processor continues its search to the right of the port at which it was interrupted[2].
The capabilities described above provide a basis for work splitting, in which one processor can be interrupted and its work divided between a number of path processors forming a new group. The extended `kappa` parallelisation primitive permits a range of scheduling strategies, considering:

- when interruption of a busy processor should occur,
- which busy processor should be interrupted,
- how many idle processors should form a new group to divide the work of an interrupted processor,

[2]other simple splitting strategies are possible, dividing both the current subtree of the busy processor and reallocating the remaining subtrees at the the current depth limit.

- and specifically for the use of BFP in the extended `kappa`, the initial and incremental depth limit.

A simple scheduler was created for an initial evaluation at Cambridge, in which an interruption occurred when the number of idle processors was three or more, and each work split was to a new group of three path processors. The processor selected for splitting was that with a current partitioning depth nearest the root of the search tree, and the incremental depth limit was double that of the interrupted busy processor. The initial partitioning depth was 1. Figure 9 gives the speedup performance for a network of $1 \ldots 30$ workstations with the Pentominoes problem, comparing the recursive splitting with oracles and kappa approach with the one-time partitioning of BFP. As with BFP, the simple implementation of oracle support with `kappa` increased the single-cpu runtimes by 9% over the standard Prolog compiler. During the execution of the Pentominoes program with the simple scheduler described, work splitting occurred approximately 60 times on the 30-processor network, averaging one split of a busy path processor every 420 milliseconds. The allocation of the new portion of work to an idle path processor required the communication of approximately 40 bytes. The comparatively low communications requirements of the SOK strategy set it apart from the majority of other parallel Prolog systems implemented, and this enables it to exploit commonly available distributed networks of general purpose workstations. Other work, such as [2, 4], has highlighted the separate attention that must be paid to the scheduling algorithms of OR-parallel systems. The scheduling interval of milliseconds or seconds in the SOK strategy rather than the microseconds of more finely grained techniques reduces the vulnerability of the system to inefficient scheduling. The new approach 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.

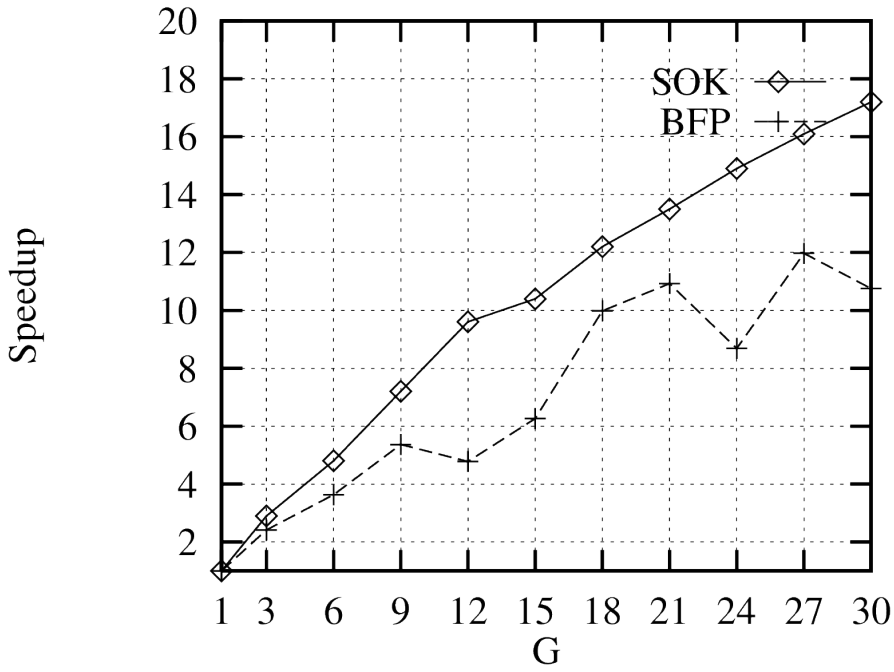

Figure 9: Speedup achieved for Pentominoes program with work splitting.

- **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.

5 Extra-logical considerations

The approach described in this paper is incompatible with the use of Prolog's extra-logical predicates *cut*, *assert* and *retract*. A *cut* discovered in the search tree of one path processor could imply a pruning of subtrees already searched or in the process of search by others. While oracles provide a basis for the communication of the discovery of a *cut*, analysis of existing Prolog code programs suggests that the rapidity with which the cuts would be discovered in the search tree would cause the overhead of communication to dominate the overall runtime [8].
Deterministic procedures produce OR-only search trees with single entry and exit points, such that oracle indexes within the procedure do not contribute to OR-parallel speedup. With oracle support removed from these procedures, the OR-parallel behaviour of the program is unaffected. This technique has been used to provide limited support for *cut* with the SOK strategy, with the requirement that any procedure containing *cut* must be deterministic.
The compiler `prologpf` developed at Cambridge implementing the SOK strategy provides support for higher-order functional programming as an alternative to the use of *cut*. No parallelisation is provided to the evaluation of functions or deterministic procedures containing *cut*. The interpretation of the functional evaluation as a tree reduction process may provide an opportunity for the extension of the oracles into the functional code, but this is an area for future work.

6 Implementation

Oracle management and partitioning support can be added to a standard Prolog program through the use of a simple extra-logical predicate, `kappa`. The sample program illustrated earlier in this paper can be transformed to use `kappa` as follows, with the predicate inserted as the first subgoal in the body of each clause:

```
% original program              % transformed program
g(U,V) :- p(U),q(V),r(U,V).     g(U,V) :- kappa(1),p(U),q(V),r(U,V).

p(1).                           p(1) :- kappa(1).
p(2).                           p(2) :- kappa(2).

q(1).                           q(1) :- kappa(1).
q(2).                           q(2) :- kappa(2).
```

Assuming an array `oracle[1...MAXORC]` to hold an oracle to be followed of length `oracle_length`, indexed by the current depth in a global variable `depth`, the search of the program can be limited to the subtree defined by that oracle with[3]:

```
kappa(I)  :-  ++depth;
              if (depth≤oracle_length)
              then if (I≠oracle[depth])
                   then fail.
              else oracle[depth] = I.
kappa(_)  :-  --depth; fail.
```

This definition of `kappa` will also accumulate the current oracle as an extension of the initial oracle given to be followed. On interruption, the current oracle can be returned by a signal handling procedure or through the use of the standard Prolog `catch` relation. Constraining the search to the right of an oracle requires the use of arithmetic comparison rather than equality, and takes advantage of the ordering of the indexes of the clauses of each source procedure.

Constraining the search of path processor `N` within a processor group of size `G` to a subset of the subtrees at a selected depth `L` can be achieved with the following definition of `kappa`:

```
kappa  :-  ++depth;
           if (depth==L)
           then { ++port_count;
                  if (port_count mod G ≠ N) then fail
                }.
kappa  :-  --depth; fail.
```

Note that partitioning from the root using `kappa` does not require oracle support and the oracle index `I` in the previous definition is not used. The one-time breadth-first partitioning described in this paper can be implemented with just this second simple definition of `kappa`. Work splitting can be implemented with a combination of both definitions given plus a procedure to return the current oracle on interruption. The treatment of `kappa` as an extra-logical predicate illustrates the principle and provides a portable implementation with an overhead typically of 10% on a single cpu. More efficient program transformations are possible, and the support provided by `kappa` could equally be embedded within the WAM call and return instructions.

Acknowledgements

This research was supported in part by the Engineering and Physical Sciences Research Council, and benefited from the supervision of William

[3]The body of each `kappa` clause should be read as a pseudo-code similar to 'C'.

Clocksin.

References

[1] K. A. M. Ali. OR-parallel execution of Prolog on a Multi-Sequential Machine. *Intl. Journal of Parallel Programming*, **15**(3):189–214, 1987.

[2] A. Beaumont. Scheduling strategies and speculative work. In *Parallel Execution of Logic Programs*, Beaumont and Gupta (Eds), Springer Verlag, 120–131, 1991.

[3] J. Briat et al. OPERA: OR-parallel Prolog System on Supernode. In *Implementations of Distributed Prolog*, Kacsuk and Wise (Eds), Wiley, 1992.

[4] R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek and R. Stevens. Scheduling OR-parallelism: an Argonne perspective. *Proc. Fifth Intl. Conf. and Symposium on Logic Programming*, MIT Press, 1590–1605, 1988.

[5] W. F. Clocksin. Principles of the DelPhi Parallel Inference Machine. *Computer Journal*, **30**:386–392, 1987.

[6] W. F. Clocksin. The DelPhi Multiprocessor Inference Machine. *Proc. 4th U.K. Conf. on Logic Programming*, K. Broda (Ed), Springer-Verlag, 189–198, 1992.

[7] T. Disz, E. Lusk and R. Overbeek. Experiments with OR-parallel logic programs. In *Proc. of the 4th Intl. Conf. on Logic Programming*, **2**:576–600, MIT Press, 1987

[8] Ian J. Lewis. PrologPF: Parallel logic and functions on the DelPhi Machine. *PhD Thesis*, Computer Lab, Cambridge University, England, 1998.

[9] E. Lusk and D. H. D. Warren and S. Haridi et al. The Aurora OR-parallel Prolog system. *New Generation Computing*, **7**:243–271, 1990.

[10] S. Saraswat. Performance evaluation of the DelPhi Machine. *PhD Thesis*, Computer Lab, Cambridge University, England, 1995.

[11] M. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *J. Auto. Reasoning*, **4**(4):353–380, 1988.

[12] D. H. D. Warren. The SRI model for OR-Parallel execution of Prolog - Abstract design and implementation issues. In *Proc. Symposium on Logic Programming*, 46–53, IEEE Computer Society Press, 1987.