Computers and Artificial Intelligence, Vol. 19, 2000, *Pageproofs!!!*

SOME ESSENTIAL SIDE-EFFECTS OF PROLOG ON A DISTRIBUTED IMPLEMENTATION*

Lourdes Araujo

Dpto. Sistemas Informáticos y Programación
Universidad Complutense de Madrid
Madrid 28040, Spain
e-mail: `lurdes@sip.ucm.es`

Abstract. This work describes an implementation of some essential side-effects of Prolog: *cut*, *findall*, and *input/output*, on a distributed memory system. The techniques designed are advantageous when applied to parallel systems based on recomputation, such as PDP (Prolog Distributed Processor), a model for Independent_AND/OR parallel execution of Prolog. Nevertheless, they are also valid for any distributed memory system. The implementation of the *cut* predicate exploits as much parallelism as possible, including the parallel computation of branches of the search tree which depend on a *cut*. However, the computation of a branch which does not depend on a *cut* is never delayed to control computations depending on a *cut*. The model proposed for the *findall* predicate reduces the communication as much as possible by distributing the control of the execution of *findall*, and takes advantage of the success paths used in the recomputation to perform the sorting of the solutions. The PDP model for the exploitation of combined parallelism has allowed a straightforward introduction of the *input/output* predicates. Results obtained with an implementation of the models on a network of workstations by means of the PVM (Parallel Virtual Machine) software package are presented.

Keywords: Logic programming, AND/OR parallelism, side-effect predicates, implementation, distributed memory.

1 INTRODUCTION

Distributed memory systems have two nice features that make them particularly attractive: they can work with a very large number of workers, and their use is widely

* Extended version of reference [7] including a detailed description of the implementation and more side-effect predicates.

extended (transputers, clusters of workstations, etc). These reasons have encouraged us to develop a parallel model of Prolog aimed to completely distributed memory systems, called PDP (Prolog Distributed Processor) [5, 6]. PDP is a recomputation-based model for the exploitation of independent_AND/OR parallelism.

The development of parallel systems to implement Prolog usually begins with the design of the part corresponding to the pure language (horn clauses) and then it is extended with side-effect predicates. PDP is not an exception. So, the purpose of this paper is to introduce some essential side-effect predicates, namely *cut* and *findall* — of which experimental results are presented — as well as *input-output*. The implementation of the database predicates is also sketched. Although the techniques proposed herein are valid for any distributed memory implementation, they turn out to be advantageous for those based on recomputation, such as PDP. Recomputation allows PDP to exploit combined parallelism as a set of independent computations (see [5, 6] for details). Thanks to this, the problem of gathering solutions inherent to AND_parallelism is overcome. Another advantage of recomputation, already discussed in [12], is that it allows to introduce side-effect predicates without sacrificing the exploitation of parallelism. What this paper will show is that side-effects can be incorporated to PDP without introducing modifications to the model; on the contrary, some of the structures of the model will be proved advantageous to implement these predicates (success path, ancestor goal, etc; see below).

For the sake of efficiency, the design of the side-effect predicates has been made so as to limit the communication among workers as much as possible.

The *cut* predicate prunes all the branches to the right of the one corresponding to the clause in which the *cut* occurs. In a sequential implementation, the same process executes every alternative clause of the procedures, exploring the search tree in a left-to-right and depth-first manner. A *cut* in the leftmost branch of a search tree is always reached before the branches on its right are explored. Therefore, the implementation of the *cut* simply consists in pruning the branches on the right hand side.

If OR_parallelism is exploited, branches of the same predicate containing *cut* may be executed by different processes. Accordingly, a process working on a non-leftmost branch in the scope of a *cut*, can find its corresponding solution before the process working on the leftmost branch reach the *cut*. Therefore, maintaining the exploitation of OR_parallelism in the presence of a *cut* requires the control of the processes which execute each non-leftmost branch in order to be able to stop them if necessary. This mechanism may be too expensive on a distributed memory system, for which stopping a process requires an exchange of messages.

In the solution proposed by Calderwood and Szeredy [9] a worker executing *cut* which is not the leftmost in the subtree prunes as much as it can, leaves the pruning of the remaining branches to a worker on its left, and then proceeds executing operations following the *cut*. When a worker detects that every left branch has failed, it will try to prune as much as it can until all branches within the scope of the *cut* are pruned. Another approach by Hausman [13] prunes at once all branches which would be pruned anyway by *cuts* with smaller scopes. Gupta and Santos

Costa [12] extend this mechanism to and-or parallel systems. Supporting *cut* in the above commented approaches requires efficient mechanisms for checking whether the current branch is the leftmost in the tree, as well as mechanisms to identify workers working on branches on the right. This may be done efficiently if the control information is accessible to all workers in a shared memory space — as it happens in systems such as Aurora [9] or MUSE [1] — but it requires a large amount of communication on a distributed memory system. Thus, these systems need a different approach. What PDP —as well as other fully distributed memory systems— needs in order to introduce side-effects is an analysis to decide how to reduce communication overhead.

Another model by Ali [3] consists in constraining the OR_parallelism exploitation to the points outside the scope of a *cut*. This "conservative" approach seems to be suitable for PDP because it avoids communication overhead, but, at the same time, many opportunities of parallelism may be lost. Accordingly, I present here a more ambitious approach. The key idea of this approach is to exploit as much parallelism as possible, but in such a way that a *safe* computation (a computation which does not depend on a *cut*) is never delayed. It is only after finishing that a safe computation pays attention to *speculative* computations (those depending on a *cut*). If a safe computation finishes with success, the speculative computations it has created are killed; otherwise, the speculative computations become safe one by one until one of them succeeds. In this way the overhead due to the control of speculative computations does not affect the safe computations. Besides, control information does not need to be accessible to all workers since each worker is responsible for pruning only the branches created by itself because of parallelism exploitation. In this way the exchange of messages is reduced, which is very convenient for a distributed memory system.

This paper also presents an implementation of the *findall* predicate, as well as its interaction with the *cut* predicate. *Findall(X, G, L)* constructs a list L consisting of all of the bindings of X for which the goal G holds. The order in the list corresponds to the order in which the proofs are found. In sequential Prolog systems, the order is determined by the depth-first, left-to-right manner in which the search tree is explored. In OR_parallel systems, solutions are found in an arbitrary order, so that recovering the sequential order for *findall* requires a special treatment. An approach by Carlsson et al. [8] implements *findall* by introducing the ability to create *sets*, defined as "ordered list". These sets, which are accessible by multiple processes, are manipulated by predicates implemented as primitives built into the OR_parallel system. In another approach by Ali [2], a worker saves its *findall* solutions in a node of a *findall tree*. Solutions are moved up from one node to another until reaching its corresponding *findall* node, following the depth-first left-to-right execution strategy. These approaches assume multiple processes sharing access to some special structure (a *set* [8] or a *findall tree* [2]). Hence, they are not appropriated for a distributed implementation. The model proposed herein reduces the communication as much as possible by distributing the control of the execution of the *findall* predicate, and takes advantage of the success paths used in the recomputation to perform the ordering of the solutions. The parallel execution model is extended to take care of the *findall* predicates appearing during recomputation so as to avoid repeating their execution. This approach for *findall* execution can be integrated with the one for the *cut* execution in a very natural way.

The side-effect mechanism [16] of the RAP model for the exploitation of AND_parallelism can be summarized as a sequencing rule according to which (i) all pure subgoals preceding an input/output (I/O) predicate subgoal must be completed successfully before the execution of the predicate, and (ii) all pure subgoal following a database predicate must not begin their execution until the predicate has been successfully completed. This proposal has a well-defined backward execution and can easily be added to an AND_parallel implementation of Prolog. Since the PDP approach for the AND_parallelism exploitation is an extension of the RAP model, the same approach for the inclusion of the I/O predicates can be adopted. The cases of OR and combined parallelism can be solved in PDP by making profit out of the structures created for parallelism exploitation.

The rest of the paper proceeds as follows: Section 2 presents an execution model overview of PDP; Section 3 describes the implementation of *cut*; Section 4 discusses the implementation of the *findall* predicate; Section 5 describes the introduction of the *input/output* predicates; Section 6 sketches the introduction of the database predicates and Section 7 draws the main conclusions of this work.

2 EXECUTION MODEL OVERVIEW

The execution model — described in full detail in [6] — has been implemented as an extension of the Warren Abstract Machine (WAM) [18], maintaining the optimizations of the sequential Prolog techniques. The goals and clauses to be executed in parallel (*parallel goals* and *parallel clauses*) are supposed to be annotated in the program, either by the compiler [10, 17] or by the user.

PDP has been devised to run on a pool of processors organized within a hierarchy of clusters, each consisting of a *scheduler* and a set of *workers*. Fig. 1 sketches the system. Schedulers are responsible for the distribution of pending work among idle workers.

Because the system is devoted to distributed memory architectures, the execution model has been designed in such a way that there are no variables in a worker defined in terms of variables that belong to another worker, thus reducing communication overhead.

2.1 AND_Parallelism Exploitation in PDP

Independent AND_parallelism is exploited by following a *fork-join* approach (see Fig. 2). Each goal g_i in a *parallel call* $(g_1, ..., g_n)$ —i.e. set of independent goals annotated to be executed in parallel— is executed on a different worker, if available. Then, the results of the execution of every goal are collected in the parent worker

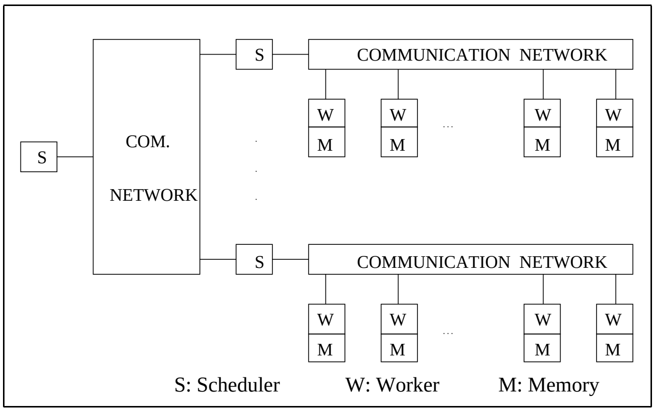

Fig. 1. PDP general architecture

(the one that found the parallel call). The execution inside each worker follows the RAP model [14] by Hermenegildo. This model extends the WAM by providing new data structures and instructions to deal with unification and backtracking performed among different workers.

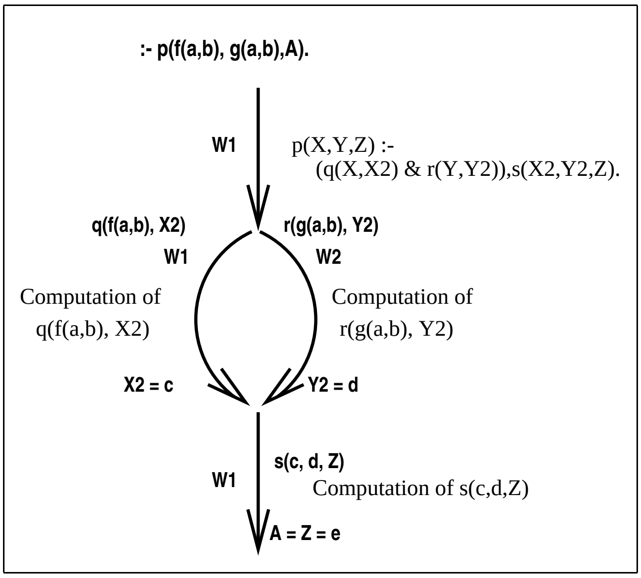

Fig. 2. Fork-join model for the exploitation of AND_parallelism. Worker W1 finds a couple of independent goals $(q(X, X2), r(Y, Y2))$ annotated with parallelism (&) and sends the second one to an idle worker W2. Because the goals are sent along with the bindings of their variables, they are computed independently, without further exchange of data. Results are joined in the worker W1 when the computations finish.

2.2 OR_Parallelism Exploitation in PDP

OR_parallelism is exploited by following a recomputation approach [4]; the environment of a worker is reconstructed by recomputing the initial goal without backtracking (see Fig. 3), following the so-called *success_path*, i.e. the sequence of clauses which have succeeded, obtained from the parent worker (the one that found the parallel clause). A worker whose computation leads to a *parallel procedure*, i.e. a procedure with OR_parallelism, can assign some of its pending work to an idle worker. If the first worker has executed the goals $(g_1, ..., g_n)$ until the appearance of OR_parallelism, and the clause it selected to solve each g_i was C_{i,j_i} (where j_i labels the clause chosen in the procedure for g_i), then the second worker computes a new branch with clauses $C_{1,j_1}, ..., C_{n-1,j_{n-1}}, C_{n,j_n+1}$.

Recomputation is not significantly more advantageous than other approaches (for instance, the copy of the worker environment [1]) as concerns OR_parallelism [4]; however, it permits to exploit OR_under_AND parallelism in a very natural way, as will be shown in the next section.

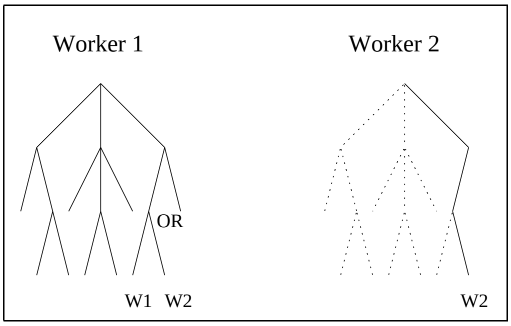

Fig. 3. Recomputation model. Worker 1 has had to explore the search tree on the left of W1 to arrive at this point, while worker 2 has only computed the shorter path (success path) until W2, which is annotated to be executed in parallel. This path to W2 is provided by worker 1.

2.3 Exploiting Combined Parallelism in PDP

The PDP approach to exploit OR_under_AND parallelism [5] has been designed to create, in an automatic and decentralized way, an independent computation for each solution.

OR_parallelism appearing under AND_parallelism is exploited by recursively splitting off the execution of OR_parallel clauses into independent branches of computation (each with a pure AND_parallel call, as illustrated in Fig. 4), thus taking advantage of the recomputation technique. This approach has two main advantages: first, it avoids storing solutions because no parallel call is waiting for the completion of the whole set of solutions, and second, the splitting off algorithm can resolve any OR_under_AND parallel call in an automatic, decentralized way, i.e. no worker has to perform the splitting, but anyone which gets idle selects, by applying the algorithm, its corresponding solution.

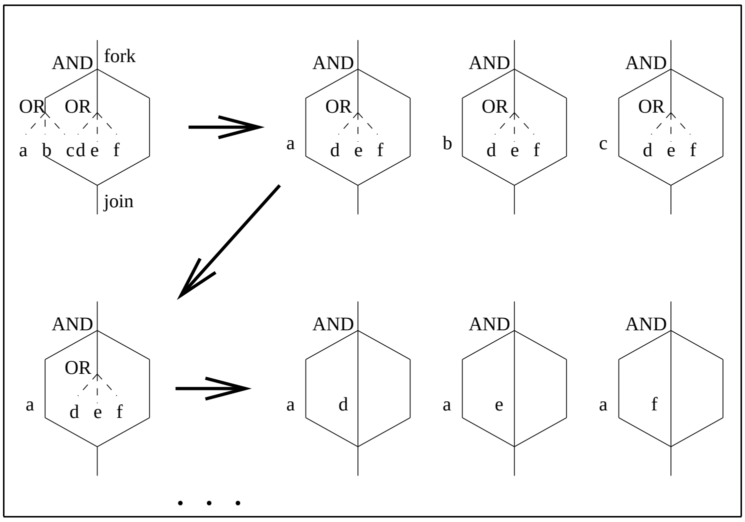

Fig. 4. Example to illustrate the combined parallelism algorithm. OR_parallelism in the first branch of a parallel call leads to independent computations of the parallel call, each of them with a different solution for the parallel branch. OR_parallelism in the remaining branches of the parallel call is exploited similarly.

The computation of a goal in PDP is called a *task*. Two types of tasks are distinguished: *OR_tasks* and *AND_tasks*. An OR_task computes solutions to the initial goal by exploring a portion of the search tree. An AND_task computes a solution to a goal which belongs to a parallel call. A task (*parent task*) exploits AND_parallelism and OR_parallelism by creating new AND_tasks and OR_tasks respectively. In this way, the parallel execution of a program defines a *task tree*. The model supports combined parallelism in a very natural way. As a result, the execution of the search tree is automatically distributed among tasks by means of a combination rule, so that no specific task is in charge of the distribution. The model is outlined as follows:

- The program execution begins as an OR_task (the root of the task tree), which performs a sequential computation until a parallel call or a parallel procedure is reached.

- The execution of a *parallel call* $(g_1, ..., g_n)$ is carried out by the creation of an *AND_task* for each independent goal g_i. These AND_tasks receive from its parent task a goal and the computed answer substitution restricted to the variables of the goal. Each AND_task computes its goal, returns the *local solution* to the parent task and finishes. The parent task waits for the answer to each independent goal and it is in charge of synchronizing the reception of those answers.

- The execution of a *parallel procedure*, i.e. a procedure with OR_parallelism, is carried out by the creation of a new *OR_task*. This receives the *success_path* of the predecessor OR_task and recomputes the initial goal following this path. After the recomputation, the execution of the next solution is computed sequentially.
- If both kinds of parallelism appear combined, parallelism is still exploited by creating the corresponding tasks. If AND_parallelism appears under OR_parallelism, the execution is performed as in the case of pure AND_parallelism, since the exploitation of OR_parallelism produces the same environment as a sequential execution.
- If OR_parallelism appears under AND_parallelism, the OR_tasks arising from an AND_task have to re-execute the parallel call in order to find new solutions to it. If this were done blindly, the result would be the simple *repetition of solutions*. To avoid this, a *combination rule* has been introduced which decides which branch is explored to solve each independent goal. Let us introduce some notation in order to define the rule.

  The *solution S to a parallel call* $(g_1, \ldots, g_n)$ is a set of local solutions to each of its goals. If a goal g_i has m_i alternative local solutions, then $s_{ij_i}, 1 \leq j_i \leq m_i$, is the $j_i th$ solution to g_i. There is an order in the set of local solutions to a goal given by a depth-first, left to right sequential execution. We also define the *ancestor goal* of an OR_task arising from an AND_task as the goal executed by this AND_task. Finally, let us give the name *predecessor OR_task* of a task T to the first OR_task in the branch going back from the task T to the root.

  The *combination rule* gives the solution which corresponds to an OR_task, in terms of the solution corresponding to the predecessor OR_task and in terms of the ancestor goal, in the following way:

  - The solution corresponding to the initial task is built from the *first* local solution to each goal in the parallel call:

    $$\{s_{1,1}, \ldots, s_{n,1}\}.$$

  - The subsequent tasks compute the solution S' which is defined in terms of the ancestor goal g_i and the solution S computed by the predecessor OR_task. Assume $S = \{s_{1,j_1}, \ldots, s_{i-1,j_{i-1}}, s_{i,j_i}, s_{i+1,j_{i+1}}, \ldots, s_{n,j_n}\}$; if $j_i < m_i$, then S' will be:

    $$S' = \{s_{1,j_1}, \ldots, s_{i-1,j_{i-1}}, s_{i,j_i+1}, s_{i+1,1}, \ldots, s_{n,1}\}.$$

    If $j_i = m_i$, then no solution remains to be explored in the branch of the task tree corresponding to the ancestor goal g_i.

> The key point of the combination rule is to fix the solution to the goals on the left of the ancestor goal and to combine them with *every* solution to the remaining goals.

The PDP approach to exploit combined parallelism — when it appears in the form OR_under_AND — is based upon the fact that the recomputation allows the AND_tasks to exploit OR_parallelism by creating OR_tasks. If an AND_task finds OR_parallelism, it creates a new OR_task to deal with the parallel clauses, and transfers to it the success_path leading to the parallel call. Notice that the AND_task has received this information only for this purpose. The new OR_task applies the combination rule in order to decide which solution is to be explored.

Each worker of PDP implements an extension of the Warren Abstract Machine (WAM) [18] which consists of the addition of new data structures and instructions to manage parallelism.

The results of the implementation of this model [6] have proved that OR_parallelism exploitation provides a linear speedup for high granularity programs. For some programs presenting both kinds of parallelism PDP achieves a greater speedup than the product of the speedups achieved by exploiting each kind of parallelism separately. The reason is that the exchange of messages required in the exploitation of AND_parallelism is avoided in PDP when OR_under_AND parallelism is exploited.

3 CUT IMPLEMENTATION

For the sake of clarity, the model is initially presented for the exploitation of OR_parallelism only, and then it is extended to a complete AND/OR_parallel execution model such as this of PDP.

3.1 Implementing *cut* in an OR_parallel Environment

In this approach all OR_parallelism is exploited. The model relies upon the distinction between *safe* tasks, i.e. computations of branches which do not depend on any *cut*, and *speculative* tasks, i.e. computations that can be pruned by pending *cuts*. The principle under which this approach has been designed is never to delay a safe task because of controlling speculative tasks.

The model can be outlined with the following points:

- OR_parallelism is exploited and all the branches of a predicate are executed simultaneously. Branches in the scope of a *cut* are executed as speculative tasks.
- If a safe task fails, the task corresponding to the execution of the next branch, in a depth-first, left-to-right order, becomes safe – by receiving a message from the previous safe task.
- If a safe task succeeds, the speculative tasks executing branches arising from the same node are killed, because at this point it is already sure that they are not going to be necessary.

- A solution reached by a speculative task is not given as an output but is stored until the task becomes safe or is killed.

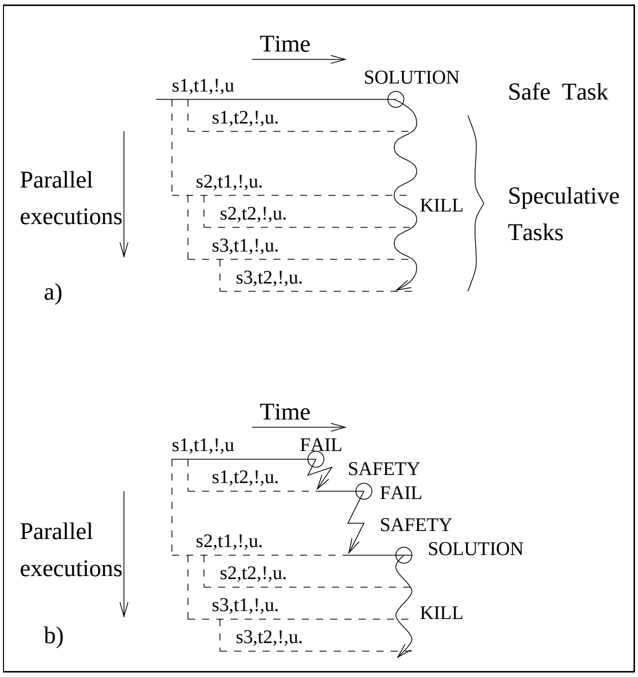

Fig. 5. Parallel implementation of *cut*

Fig. 5 shows the scheme of two possible cases in the execution of the goal p in the following program:

```
p :- s, t, !, u.

s :- s1.            t :- t1.
s :- s2.            t :- t2.
s :- s3.            u. s1. s2. s3. t1. t2.
```

If the computation corresponding to $s1$ and $t1$ does not fail, the remaining computations are unnecessary because of the *cut*. In Fig. 5a) the scheme shows an execution in which the first task, which is safe, gives the solution and then the remaining tasks, which are speculative, are killed. Fig. 5b) shows an example in which the first task fails, and then informs the following speculative task of its new safe status. The process is repeated until a solution is reached.

3.2 *cut* Implementation in PDP

The appearance of a *cut* in an AND_parallel computation [14] causes the elimination of the alternative solutions of the goals in the parallel call affected by the *cut* (see Fig. 6).

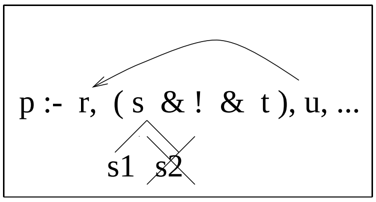

Fig. 6. Cut in an AND_parallel execution. The cut behind the goal s prunes the branch of $s2$ after $s1$ succeeds. Accordingly, if u fails backtracking will be performed on a previous goal to the parallel call because t has no alternatives.

The combined parallel model of PDP transforms the OR_under_AND parallelism in AND_under_OR parallelism, and each solution is explored by an independent task by means of the recomputation. This requires giving to the AND_tasks the capability of creating OR_tasks. Thus, the implementation of *cut* in PDP leads to extend the distinction between safe and speculative tasks as follows:

- **speculative AND_task:**
  An AND_task is *speculative* in these two cases: (a) if the task executes a goal on the left of a *cut* in a parallel call, or (b) if the task arises from another speculative task. Since the solution to the goals on the left of a *cut* in a parallel call are fixed to the first solution encountered, in case (a) any speculative OR_task that the AND_task might create is killed as soon as the solution to the goal is reached, or it is made safe as soon as the AND_task fails.

  Notice that, in spite of being speculative, an AND_task always computes the first solution to its goal and gives it to its parent task. What the speculative character determines is that every task it creates is also speculative.

- **speculative OR_task:**
  Any OR_task arising from a speculative AND_task or from another speculative OR_task is also *speculative*, and exploits parallelism by means of speculative tasks. The solution obtained by a speculative OR_task is stored until the task becomes safe or is killed.

Fig. 7b) shows a PDP execution example corresponding to the program in Fig. 7a).

The execution begins with the OR_task which computes the solution (p_1, q_1). New AND_tasks are created for the execution of the goals in the parallel call, the task which executes p, on the left of the *cut*, being speculative. Since the goal

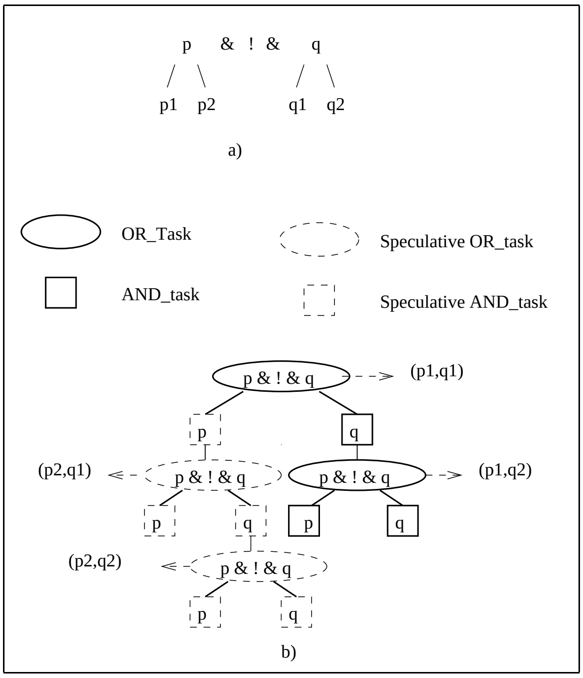

Fig. 7. Cut in a program with combined parallelism

p presents OR_parallelism, a new speculative OR_task is created to compute the solution corresponding to (p_2, q_1). Again the parallel call is executed by means of AND_tasks, both speculative in this case because they arise from a speculative OR_task. The execution of the speculative goal q presents OR_parallelism which is exploited by a speculative OR_task to compute the solution (p_2, q_2). These speculative OR_tasks compute the assigned solution until either being killed by the first AND_task computing p if the solution corresponding to p_1 is reached, or becoming safe if the computation of p_1 fails. This process is performed recursively, in such a way that if p had more than two alternatives, the third one would only become safe provided the second one had become safe and failed, and so on. If the solutions of the speculative OR_tasks are reached before being killed or becoming safe, they are stored waiting for one or another event. Notice that in the execution of the solution (p_1, q_2) there are no speculative tasks because, according to the PDP model, the solution to be explored for the goals (p) on the left of the one whose parallelism is being exploited (q) are fixed.

3.3 Scheduling Policy

In PDP, the schedulers are responsible for the distribution of pending work among idle workers. A worker will always be in one out of three states: *idle* (without work), *busy* (working) or *offering* (with pending work). The scheduler is informed about idle and offering workers and it selects for each idle worker the offering worker to be requested for work. The scheduling policy takes into account the distinction between safe and speculative work in such a way that safe jobs are preferred, and if only speculative jobs exist, the jobs corresponding to leftmost branches are preferred (they are the least speculative).

3.4 Evaluation

PDP, which was initially implemented on a transputer network, has now been ported to a workstation network using PVM (*Parallel Virtual Machine*) [11], a software package that allows a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. Although the model is valid for any distributed memory system the network of workstations are more widely used than other platforms, such as transputer networks, and they have better tools for software development. Nevertheless, the execution times obtained are not optimal since the resources have to be shared with other processes. The current results have been obtained with 12 processes running on three SUN SPARC 1 workstations (20 MHz).

The above presented scheme has been compared with the other one in which the exploitation of parallelism is constrained to the part of the program outside the scope of a *cut* (the method used in [3]). The programs used as benchmarks are a synthetic program, which presents coarse grain parallelism and it is very convenient for our system; the *mastermind* program, with eight digits to choose and different secret codes — mm1([1,5,0,4,3]), mm2([2,2,2,2,2]), mm3([3,3,4,5,1]), mm4([5,4,3,2,0]) —; the *number* program, previously used by Hausman [13], with five digits to choose and different queries — number1(3,62), number2(3,65), number3(3,56), number4(3,88) —; and the Chat-80 natural language query system, with the query *db5* used by Hausman (db5 *20 means that the query was run twenty times). Table 1 shows the results for these programs. In order to give an indication of the speed of the PDP with respect other Prolog systems, the first column in the table presents the execution times of running the benchmarks with SICStus Prolog (version 3), which are similar to those of sequential PDP.

The speedup obtained for these programs with the unconstrained approach is only slightly lower than the one obtained by Hausman [13] with 4 workers in the Aurora system, which presents some shared memory. According to the results the unconstrained approach seems to be better. For some programs, such as number, the constrained approach prevents the parallelism exploitation. Nevertheless, there can be programs with finer grain parallelism whose exploitation does not compensate the overhead introduced to deal with the *cut* in the unconstrained approach. Of course, both methods can be combined since parallelism can be properly annotated [6] so as to exploit parallelism only if it is advantageous.

program	Sequential		Parallel			
	Sicstus	PDP	Constrained		PDP	
	T (s.)	T (s.)	T (s.)	Sp	T (s.)	Sp
synthetic	7.03	7.10	7.10	1	2.81	2.53
mm1	62.98	63.80	48.72	1.31	35.64	1.79
mm2	25.28	25.16	21.50	1.17	16.55	1.52
mm3	144.45	151.45	101.64	1.49	55.68	2.72
mm4	149.34	165.94	109.89	1.51	57.79	2.87
number1	23.70	24.07	24.05	1	20.23	1.19
number2	45.76	49.32	49.41	1	28.51	1.73
number3	37.82	38.11	37.72	1.01	34.50	1.10
number4	21.10	21.54	21.60	1	16.20	1.33
db5*20	3.74	3.70	3.71	1	2.74	1.35

Table 1. Comparison of approaches for implementing *cut*. Seq. stands for sequential, T for Time and Sp for speedup.

4 FINDALL IMPLEMENTATION

The PDP implementation of *findall* introduces some differences in the exploitation of OR_parallelism. One of them is the fact that a solution corresponding to a *findall* predicate does not have to be sent to the output process, but to the one executing the *findall*. A second key point is to detect when every solution to the *findall* predicate has been collected. Since the PDP execution model creates independent computations for the execution of every solution, additional mechanisms are required in order to detect the end of every task taking part in the execution of a *findall*.

Let us call *F_task* the task executing the *findall* predicate. This task may be either an OR_task or an AND_task. Thus, the extension of the model to implement *findall* introduces two new types of tasks:

- **Findall_OR_task:**
  This task computes a solution to the goal of the *findall* predicate (*findall goal*) instead of the initial goal. It receives a success path which starts at the findall goal and leads to a new solution to it. The obtained solution is sent to the F_task instead of the input_output process, as it would be the case of a normal OR_task. If this task finds parallelism it exploits it by further findall_tasks.

- **Findall_AND_task:**
  This task is similar to an AND_task, except for the fact that it exploits parallelism by further findall_tasks. As in the case of the implementation of *cut*, AND_tasks are merely transmitters of the character (speculative or findall) they have. It is in the OR_tasks that this character is translated into an actually different behavior.

The *findall* execution approach is outlined in the following points:

- A task executing a *findall* predicate enters a new execution mode, *findall_mode*, in which its behavior is adapted to the execution of *findall*.
- The OR_parallelism and the AND_parallelism of the findall goal is exploited by means of findall_OR_tasks and findall_AND_tasks respectively.
- In findall_mode a task receives solutions until the completion of the *findall* predicate. Two strategies have been investigated for this detection, which will be discussed later.
- Each solution is received along with its computed success path, which allows sorting the solutions in a special storing structure.
- After receiving every solution to the findall goal, a list with the solutions is built in the *Heap* of the WAM. This list is also stored in an auxiliary buffer connected to the success path, which avoids computing *findall* again in later recomputations (see below). Finally the computation mode is changed to normal mode. In the case of nested findall predicates the F_task has to take into account that it can be in findall mode itself, and if so the list of solutions it has computed has to be sent in its turn to its F_task.

Fig. 8 shows a scheme of the mechanism. First of all, the work is distributed among idle workers. This is done by creating a findall_OR_task for each solution. The results obtained are sent to the parent task. The F_task is informed of the failure of any task taking part in the execution of *findall*. The *findall* ends as soon as every failure has been reported. In order to detect when this has happened two different strategies have been devised which will be discussed later on.

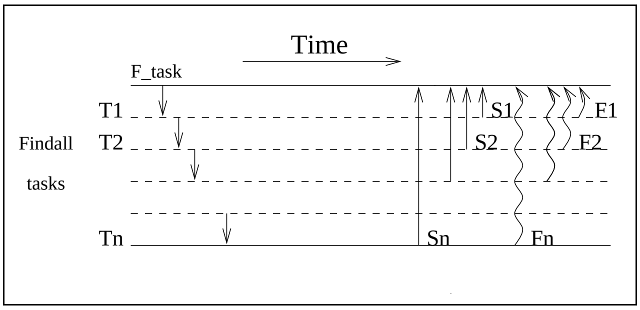

Fig. 8. The mechanism to implement *findall*. T_n, S_n and F_n stand for the n-th task, solution and failure respectively.

4.1 Sorting the Solutions

The *findall* predicate collects the solutions to a goal in a list. If parallelism is being exploited, the solutions are sorted in the list as they have been obtained. This ordering may have changed in different executions of the same program, and it may be unconvenient for the user to obtain lists with different sortings in subsequent runs of the program. The mechanism presented in the previous section can be extended to sort the solutions. This is done by associating to each solution its success path. Success paths are sorted by lexicographical order, assuming the usual left-to-right order at the nodes of the search tree.

4.2 Extension of the Abstract Machine

The implementation of the *findall* predicate following the above described scheme requires a mechanism to store and sort the collected solutions. To this purpose two auxiliary stacks and a linked list are created. The first stack (*auxiliary solution stack*) stores the bindings of the variable appearing in the *findall* predicate. The second stack (*auxiliary success stack*) stores the corresponding success paths starting from the *findall*. Finally, the nodes of the list store the addresses of these two data in their respective stacks. (See Fig. 9.)

Whenever a new solution is found it is placed in the list in its proper order, which is determined by means of the information stored in the auxiliary success stack. The solution and its success path are pushed in their respective stacks. After completion of the *findall* the corresponding term is created in the *Heap*.

Although the existing information in the stacks is not necessary any more, it can be used in order to introduce an optimization. Notice that recomputation is not advantageous whenever a *findall* appears within the success path, for *findall* has to compute *every* solution to its goal. However, the recomputation of the *findall* can be avoided if a sorted copy of the list of solutions is linked to the success path.

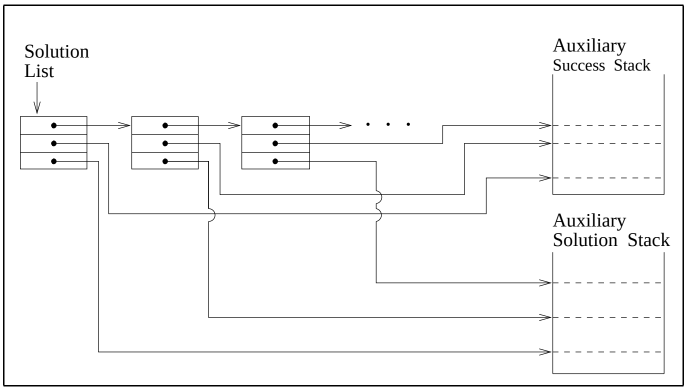

Fig. 9. Implementation of *findall*

4.3 Detection of *Findall* Completion

The parent task needs to know when all the solutions to *findall* have been found. This happens when every task taking part in the computation has failed. Two strategies have been checked to detect this event.

- *Strategy* 1*: Failure is reported to the parent task* (Fig. 10a).
  Each task taking part in the execution of *findall* collects the failures of its offspring tasks. If the task fails before all its offspring tasks, it informs the remaining of them that their new parent task is its own parent task. Besides, it informs its parent task, along with the reporting of its failure, of its "adopted" offspring tasks.

- *Strategy* 2*: Failure is reported to the F_task* (Fig. 10b).
  The tasks taking part in the execution of *findall* report their failure directly to the F_task. In this strategy the F_task needs to knows how many tasks participate in the execution of *findall*. Therefore, each task reporting a solution or a failure must also inform about how many tasks it created.

In both cases the possible reversing in the ordering of creation and termination reports is considered, and a task receiving more termination reports than expected at a given moment does not consider it a mistake because it will eventually receive the corresponding reports of creation.

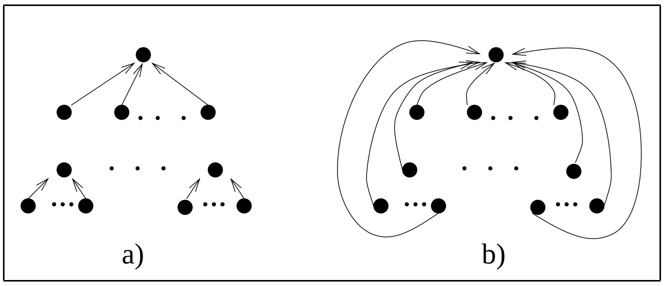

Fig. 10. Detecting *findall* completion

The number of message exchanges is similar with both strategies. However, they introduce some differences which make this point relevant when there is a large number of tasks taking part in the computation of *findall*. The implementation of the first strategy has the advantage of decentralizing the process, but the disadvantage of introducing more exchange of messages whenever a task fails before all its offspring tasks. For the second strategy it is the other way around.

Table 2 shows the results of implementing each of these strategies, as well as the time to collect all solutions in a 'program, fail' manner. The latter allows to estimate the overhead introduced by the findall mechanism (around 15% in both approaches). The programs used as benchmarks are the *queen* program; the *mastermind* program,

program	sequential (prog, fail)	parallel (prog, fail)		findall strategy a		findall strategy b	
	T (s.)	T (s.)	Sp	T (s.)	Sp	T (s.)	Sp
queen(6)	1.7	1.37	1.24	1.5	1.13	1.5	1.13
queen(8)	39.0	14.68	2.65	17.0	2.29	15.40	2.53
mm5	379.89	102.90	3.69	118.13	3.21	116.92	3.25
mm6	356.99	99.65	3.58	110.14	3.24	109.85	3.25
mm7	388.31	112.78	3.44	121.10	3.20	119.48	3.25
number5	58.97	17.33	3.40	23.57	2.50	23.02	2.56
number6	82.17	25.97	3.16	32.28	2.54	31.90	2.57
number7	138.60	43.60	3.18	54.84	2.53	54.09	2.56

Table 2. Comparison of the strategies for the detection of the *findall* completion. T stands for Time and Sp for speedup.

with four digits to choose and different secret codes – mm5([1,1,2]), mm6([0,3,1]), mm7([3,3,2]) –; and the *number* program, with five digits to choose and different queries – number5(3,15), number6(3,5), number7(3,3). Results show slightly shorter execution times for strategy b. They seem to indicate that the difference increases with the program size and the number of solutions to collect. The reason is the smaller number of messages exchanged with strategy b. The number of messages exchanged with both strategies is the same unless a parent task fails before their offspring tasks. However, in executions of goals with a large number of solutions this situation appears frequently and that makes the difference. On the other hand, results show that the centralization of the collection of solutions does not introduce much overhead even for goals with a large number of solutions, such as *queen(*8*)* or *number*.

4.4 Cut and Findall

The interaction between these predicates is managed as a natural extension of the implementation of the *cut* predicate. If a *cut* appears in the execution of a *findall* predicate the findall tasks are *speculative* and their computed solutions are not sent to the F_task until the findall_task becomes *safe*. A different strategy could be to prune the solutions corresponding to speculative tasks in the F_task, but this strategy forces a delay in the F_task and introduces a larger number of message exchange, thus making it inappropriate for a distributed implementation.‘

5 INPUT/OUTPUT PREDICATES

As mentioned in the introduction, PDP deals with I/O predicates arising in AND_ parallelism exploitation in the same way as the RAP model [16]. For the exploitation of OR_parallelism, the execution of the I/O predicates in a recomputation model requires to avoid recomputing these predicates. In the case of the *read* predicate this amounts to include in the success path the data read in the first execution of the predicate.

The previous approaches have to be extended for the OR_under_AND parallel model. In a sequential execution backtracking may cause multiple executions of an I/O predicate. This will happen whenever there is a predicate with alternative clauses on the left of the I/O predicate. PDP deals easily with this problem thanks to its mechanism to exploit combined parallelism. As already pointed out, the way in which PDP deals with an OR_under_AND parallel clause is to create independent computations for each combination of solutions to the parallel call. Each of these computations has information on the precise goal whose OR_parallel exploitation has caused it, which is its *ancestor goal*. Therefore, an I/O predicate will be reexecuted only if it is on the right of this goal. For instance, let us consider the following program:

```
p :- (s & t & write & u).
u :- u1.
u :- u2.
u :- u3.
```

The OR_parallelism of goal u leads to create tasks to solve $s\&t\&write\&u1$ (T1), $s\&t\&write\&u2$ (T2), and $s\&t\&write\&u3$ (T3). The ancestor goal of T1 is p, and thus the *write* predicate, to its right, will be executed. On the other hand, since T2 and T3 are created because of u's parallelism, u is their ancestor goal, and therefore the *write* predicate to its left will not be reexecuted. Again, the general PDP mechanism for the exploitation of parallelism allows to introduce side-effects without additional computation mechanisms.

6 DATABASE PREDICATES

The updating of the program database on every processor may be very expensive on a distributed memory system. Besides, the implementation of the general *assert* requires a full compiler and retention of the compile-time environment during execution. Therefore, it seems reasonable to avoid the exploitation of parallelism in the presence of these predicates. However, many programs only assert clauses for predicates whose functor, arity, and argument types are known at compile time (simple clauses). As proposed in [15], the assertion of this kind of clauses may be optimized by generating at compile time both a clause template for each predicate and a clause-specific routine (demon). At run time the demon copies and fills in the template for each clause asserted. This restricted assertion could be implemented in PDP as well.

7 CONCLUSIONS

This paper has presented an implementation of some important side-effects in PDP. The implementation reveals that the parallelism exploitation model of PDP turns out to be advantageous to include the side-effects to the extent that the model need not be altered in any way.

An "unconstrained" *cut* implementation model, which exploits all parallelism appearing in the program, has been compared with an approach, already proposed in the literature, which constrains OR_parallelism exploitation to those parts of the program outside the scope of any *cut*. In the unconstrained model idle workers can carry out speculative computations, which leads to an increase in the performance if they eventually become safe. Anyhow, the model is based upon the principle of never to delay safe computations because of controlling speculative ones. The results obtained implementing both models on a network of workstations reveal that the unconstrained approach is clearly advantageous.

The *findall* predicate has also been implemented in such a way that the control of the execution is decentralized. Two strategies have been tried for detecting that every solution has been collected. It turns out that the best strategy is that in which failures are reported directly to the task executing the *findall* predicate instead of the parent tasks. This approach for *findall* execution can be combined with the one for *cut* execution in a very natural way.

Input/output predicates have also been designed for PDP. The side-effects mechanism of the RAP model has been adopted in PDP for the exploitation of AND_parallelism. The PDP model for the exploitation of combined parallelism has allowed to introduce the input/output predicates without further extensions.

Finally, it has been suggested that the model by Mills and Buettner for the exploitation of the database predicates can be adapted to PDP as well.

REFERENCES

[1] Ali, K. A. M.—Karlsson, R.: The Muse Approach to Or-Parallel Prolog. International Journal of Parallel Programming, Vol. 19, 1990, No. 2, pp. 129–162.

[2] Ali, K. A. M.—Karlsson, R.: A Novel Method for Parallel Implementation of findall. Proceedings of the International Conference on Logic Programming, MIT Press 1989, pp. 235–245.

[3] Ali, K. A. M.: A Method for Implementing Cut in Parallel Execution of Prolog. Research Report R87001, Swedish Institute of Computer Science 1987.

[4] Araujo, L.—Ruz, J.J.: OR-Parallel Execution of Prolog on a Transputer-based System. Transputers and Occam Research: New Directions. IOS Press 1993, pp. 167–181.

[5] Araujo, L.—Ruz, J.J.: PDP: Prolog Distributed Processor for Independent_AND\OR Parallel Execution of Prolog. Proceedings of the International Conference on Logic Programming, MIT Press 1994, pp. 142–156.

[6] Araujo, L.—Ruz, J.J.: Parallel Prolog System for Distributed memory. Journal Logic Programming, Vol. 33, 1997, No. 1, pp. 49–79.

[7] Araujo, L.: Towards Full Prolog on a Distributed Architecture. Proceedings of Euro-Par'97, Springer-Verlag 1997, LNCS 1300, pp. 1173–1180.

[8] Carlsson, M.—Danhof, K.—Overbeek, R.: A Simplified Approach to the Implementation of AND-parallelism in an OR-parallel Environment. Proceedings of the International Conference on Logic Programming, MIT Press 1988, pp. 1565–1577.

[9] Calderwood A.—Szeredi, P.: Scheduling Or-parallelism in Aurora — the Manchester scheduler. Proceedings of the International Conference on Logic Programming, MIT Press 1989, pp. 419–435.

[10] Chang, J. H.—Despain, A.—Degroot, D.: AND-parallelism of logic programs based on a static dependency analysis. Proceedings of Spring Compcon, IEEE Press 1985, pp. 218–225.

[11] Geist, A.—Beguelin, A.—Dongarra, J.—Jiang, W.—Manchek, R.—Sunderam, V.: PVM: Parallel Virtual Machine. MIT Press 1994.

[12] Gupta, G.—Santos Costa, V.: Cut and Side-effects in AND-OR Parallel Prolog. The Journal of Logic Programming Vol. 27, 1996, No. 1, pp. 45–71.

[13] Hausman, B.: Pruning and Speculative Work in OR-Parallel PROLOG. PhD thesis, Swedish Institute of Computer Science 1990.

[14] Hermenegildo, M.: An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Program in Parallel. PhD thesis, University of Texas at Austin 1986.

[15] Mills, J. W.—Buettner, K. A.: Assertive Demons. Proceedings of the International Conference on Logic Programming, MIT Press 1988, pp. 1403–1414.

[16] Muthukumar, K.—Hermenegildo, M.: Complete and Efficient Methods for Supporting Side-effects in Independent\Restricted And-parallelism. Proceedings of the International Conference on Logic Programming, MIT Press 1989, pp. 80–97.

[17] Muthukumar, K.—Hermenegildo, M.: Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. North American Conference on Logic Programming, MIT Press 1989, pp. 166–185.

[18] Warren, D. H. D.: An Abstract Prolog Instruction Set. Technical Note 309, SRI International 1983.

Manuscript received 9 December 1997; revised 9 February 1999

Lourdes Araujo received her B.S. degree in physics and Ph.D. in computer science in 1987 and 1994, respectively, both at the Complutense University of Madrid. After spending three years working on the design and fabrication of communication networks, she joined the Complutense University in 1990, where she is currently an Associate Professor of Computer Science. Her doctoral research focused on a parallel implementation of Prolog. Her research interests include logic and constraint programming as well as parallel computer architectures.