Towards Full Prolog on a Distributed Architecture [1]

Lourdes Araujo
Fac. Matemáticas (Dpto. Sistemas Informáticos y Prog.)
Universidad Complutense de Madrid
Madrid 28040, Spain
lurdes@dia.ucm.es

Abstract. This paper presents an implementation of some essential side-effects of Prolog: *cut* and *findall*, on a distributed memory system. Although the techniques proposed herein are valid for any distributed memory implementation, they are advantageous in those based on recomputation, such as PDP (Prolog Distributed Processor), a model for Independent_AND/OR parallel execution of Prolog. The key idea to implement the *cut* predicate is to exploit as much parallelism as possible, but in such a way that the computation of a branch of the search tree which cannot be pruned by a *cut* is never delayed to control computations depending on a *cut*, (i.e. to analyze the pruning of the branch of these computations and to kill them). The model proposed for the *findall* predicate reduces the communication as much as possible.

1 Introduction

PDP (Prolog Distributed Processor) [4, 5] 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*. Although the techniques proposed herein are valid for any distributed memory implementation, they turn out to be advantageous in those based on recomputation, such as PDP. Recomputation allows PDP to exploit combined parallelism as a set of independent computations (see [4, 5] for details). Thanks to this, the problem of gathering solutions inherent to AND_parallelism is overcome. Another advantage of recomputation, already discussed in the literature [9], is that it allows to introduce side-effect predicates without sacrificing parallelism exploitation. What this paper will show is that side-effects can be incorporated to PDP without introducing modifications to the model.

The *cut* predicate prunes all the branches on the right of the one corresponding to the clause in which the *cut* occurs. If OR_parallelism is exploited, branches of the same predicate containing a *cut* may be executed by different processes. Accordingly, a process working on a branch which is not the leftmost in the search tree can find its corresponding solution before the process working on the leftmost branch reaches the *cut*. Therefore, maintaining the exploitation of OR_parallelism in the presence of a *cut* requires the control of the processes

[1] Supported by the projects TIC95-0433.

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. A number of solutions have been proposed for parallel systems [7, 10, 9]. However, they require a large amount of communication on a distributed memory system. Another model by Ali [2] consists in constraining the OR_parallelism exploitation to the cases outside the scope of a *cut*. This "constrained" approach seems to be suitable for PDP because it avoids communications overhead, but, at the same time, many opportunities of parallelism may be lost. Accordingly, a more ambitious approach has been devised. 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 can not be pruned by a *cut*) is never delayed.

This paper also presents an implementation of the *findall* predicate. As it is well-known, *findall(X, G, L)* constructs a list L consisting of all of the bindings of X for which the goal G holds. Some solutions have been proposed for this predicate [6, 1]. These approaches assume multiple processes sharing access to some special structure (a *set* [6] or a *findall tree* [1]). Hence, they are not appropriate 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.

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; and section 5 draws the main conclusions of this work.

2 Execution Model Overview

The execution model —described in full detail in Ref. [5]— has been implemented as an extension of the Warren Abstract Machine (WAM) [13]. 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*. Schedulers are responsible for the distribution of pending work among idle workers.

The goals and clauses to be executed in parallel (*parallel goals* and *parallel clauses*) are annotated in the program. Independent AND_parallelism is exploited by following a *fork-join* approach, which is an extension for distributed memory systems of the one followed in the RAP model [11]. OR_parallelism is exploited by following a recomputation approach [3]; a processor environment is reconstructed by recomputing the query without backtracking, following the socalled *success_path*, i.e. the sequence of clauses which have succeeded until the last choicepoint with pending parallel alternatives, obtained from the parent processor (the one finding the parallel clause). Recomputation allows the exploitation of OR_under_AND parallelism in a very natural way. The PDP approach to exploit OR_under_AND parallelism [4] is designed to create, in an automatic and decentralized way, an independent computation for each solution.

The computation of a goal in PDP is called a *task*. Two types of tasks are distinguished: *OR_tasks* and *AND_tasks*. AND_parallelism and OR_parallelism are exploited by AND_tasks and OR_tasks respectively. They are created by the task finding the annotation of parallelism, which is their *parent task*. An OR_task computes solutions to the query by exploring a portion of the search tree. An AND_task computes a solution to a goal which belongs to a parallel call and gives the result to its parent task. 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, 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 clause is reached.
- The execution of a *parallel call* is carried out by the creation of an *AND_task* for each independent goal. 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 clauses annotated 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 query following this path. After the recomputation, the execution continues as usual.
- 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, it has been introduced a rule, called *combination rule* which decides which branch is explored to solve each independent goal. The *ancestor goal* of an OR_task arising from an AND_task is defined as the goal executed by this AND_task. The combination rule fixes the solution to the goals on the left of the ancestor goal and combines them with *every* solution of the remaining goals.

The results of the implementation of this model [5] 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

The model relies upon the distinction between *safe* tasks, i.e. computations of branches which can not be pruned by *cut*, and *speculative* tasks, i.e. computations that can be pruned by *cuts*. The principle under which this approach has been designed is never to delay a safe task because of controlling speculative tasks, i.e. because of analyzing if these tasks are computing a branch which has been pruned by a cut and killing them. 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*, i.e. which can be pruned by 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.
- A solution reached by a speculative task is not given as an output but it is stored until the task becomes safe or it is killed.

Figure 1 shows the scheme of a possible case 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.
```

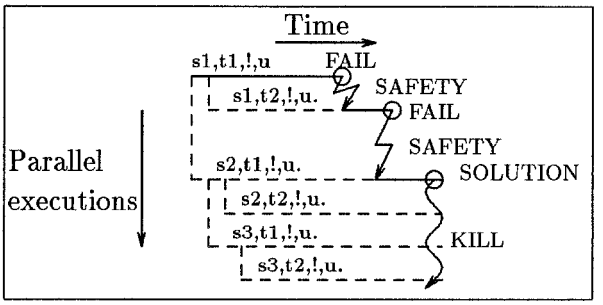

Fig. 1. Parallel implementation of *cut*.

If the computation corresponding to $s1$ and $t1$ does not fail, the remaining computations (speculative) would be useless because of the *cut* and they would be killed. Figure 1 shows the case in which the first task fails when executing

$t1$, and then informs the following speculative task $(s1, t2, !, u)$ about its new safe status. This task also fails and transforms the next task $(s2, t1, !, u)$ into *safe*. This task does not fail and when the solution is reached the remaining speculative computation are killed.

The implementation of *cut* in PDP leads to distinguish 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, for instance b in $a, (b\&!\&c)$, 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 the 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 it is killed.

PDP, which was initially implemented on a transputer network, has now been ported to a workstation network using PVM (*Parallel Virtual Machine*) [8], a software package that allows a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. Nevertheless, the execution times obtained are not optimal since the resources have been 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 ref. [2]). The benchmarks are a synthetic program, which presents coarse grain parallelism; 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 [10], 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. The speedup obtained for these programs with the unconstrained approach is only slightly lower then the one obtained by Hausman [10] 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 impedes the parallelism exploitation.

program	Sequential	Constrained	PDP
synthetic	7.10	7.10(1)	2.81(2.53)
mm1	63.80	48.72(1.31)	35.64(1.79)
mm2	25.16	21.50(1.17)	16.55(1.52)
mm3	151.45	101.64(1.49)	55.68(2.72)
mm4	165.94	109.89(1.51)	57.79(2.87)
number1	24.07	24.05(1)	20.23(1.19)
number2	49.32	49.41(1)	28.51(1.73)
number3	38.11	37.72(1.01)	34.50(1.10)
number4	21.54	21.60(1)	16.20(1.33)
db5*20	3.7	3.7(1)	2.74(1.35)

Table 1. Comparison of approaches for implementing *cut* (time in seconds and speedups in brackets.)

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 all solutions to the *findall* predicate have been collected.

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_tasks* and *findall_AND_tasks*. A findall_OR_task computes a solution to the goal of the *findall* predicate (*findall goal*) instead to the query. 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. A findall_AND_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. Finally the computation mode is changed to normal mode.

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. In *Strategy 1* 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. In *Strategy 2* failure is directly reported 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. 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

program	sequential (prog, fail)	parallel (prog, fail)	findall strategy a	findall strategy b
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 (time in seconds and speedups in brackets.)

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, 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 by both strategies is the same unless a parent task fails before their offspring tasks.

5 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*. 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 when all solutions have been collected. It turned 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.

References

1. Ali, K. A. M., Karlsson, R. *A Novel Method for Parallel Implementation of findall.* Proc. Int. Conf. on Logic Programming (1989), pp. 235-245.
2. Ali, K.A.M. *A Method for Implementing Cut in Parallel Execution of Prolog.* Research Report SICS R87001 (1987).
3. 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.
4. Araujo, L., Ruz, J.J. *PDP: Prolog Distributed Processor for Independent_AND\OR Parallel Execution of Prolog.* Proc. Int. Conf. on Logic Programming (1994), pp. 142-156.
5. Araujo, L., Ruz, J.J. *A Parallel Prolog System for Distributed memory.* The Journal of Logic Programming, 33(1), (1997), pp. 49-79.
6. Carlsson, M., Danhof, K., Overbeek, R. *A Simplified Approach to the Implementation of AND-parallelism in an OR-parallel Environment.* Proc. Int. Conf. on Logic Programming (1988), pp. 1565-1577.
7. Calderwood A., Szeredi, P. *Scheduling Or-parallelism in Aurora - the Manchester scheduler* Proc. Int. Conf. on Logic Programming (1989), pp. 419-435.
8. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V. *PVM: Parallel Virtual Machine* MIT Press (1994).
9. Gupta, G., Santos Costa, V., *Cuts and Side-effects in AND-OR Parallel Prolog.* The Journal of Logic Programming 27(1), (1996), pp. 45-71.
10. Hausman, B. *Pruning and Speculative Work in OR-Parallel PROLOG.* PhD thesis, SICS (1990).
11. Hermenegildo, M., *An abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Program in Parallel.* PhD thesis, U. of Texas at Austin (1986).
12. Muthukumar, K., Hermenegildo, M., *Complete and Efficient Methods for Supporting Side-effects in Independentbackslash Restricted And-parallelism* Proc. Int. Conf. on Logic Programming (1989), pp. 80-97.
13. Warren, D.H.D., *An Abstract Prolog Instruction Set.* Tech. Note 309, SRI International, (1983).