PDP: Prolog Distributed Processor for Independent_AND\OR Parallel Execution of Prolog [1]

Lourdes Araujo
Jose J. Ruz
Dpto.Informática y Automática
Universidad Complutense de Madrid
Madrid 28040, Spain
{lurdes, jjruz}@dia.ucm.es

Abstract

PDP (Prolog Distributed Processor) is a multisequential system for Independent_AND\OR parallel execution of Prolog. The system is composed of a set of workers controlled hierarchically. Each worker operates on its own private memory and interprocessor communication is performed only by the passing of messages. Independent AND_parallelism is exploited following a *fork-join* scheme and OR_parallelism is exploited following a multisequential approach. Both kinds of parallelism are implemented using closed environments. To exploit OR_parallelism, the parent worker environment is reconstructed in a new worker by *recomputing* the initial goal without backtracking, following the *success path* obtained from the parent worker. PDP deals with OR_under_AND parallelism producing the solutions of a set of parallel goals in a distributed way, that is, creating a new task for each element of the cross product. This approach has the advantage of avoiding both storing partial solutions and synchronizing workers, resulting in a largely increased performance. PDP has been implemented on a transputer network and performance results show that PDP introduces very little overhead into sequential programs, and provides a high speedup for coarse grain parallel programs.

1 Introduction

There are a considerable number of approaches proposed for the parallel execution of Prolog, dealing with OR_parallelism [1, 4, 5, 2], Independent AND_parallelism [10, 12, 14, 7], and recently combining AND-OR parallelism [19, 3, 11]. Since most of these approaches are implemented on systems with totally or partially shared memory, we are interested in investigating the possibilities of a completely distributed model of processing, that eliminates the shared memory contention.

[1] Sopported by the Prontic project TIC92-0793-C02-01.

In this paper we present the Prolog Distributed Processor, a multisequential system supporting both Independent_AND and OR_parallelism. In order to reduce the communications overhead, PDP has been designed with a hierarchic control. PDP is composed of a set of clusters, each of them consisting of a *controller* and a set of *workers*. Controllers are responsible for the distribution of pending work among idle workers. When every worker in a cluster is busy, the work may be sent to other clusters. Each worker operates on its own private memory and interprocessor communication is performed only by the passing of messages. In order to reduce communication overhead, each worker follows a *closed environment approach* [6], that is, there are no variables in a worker defined in terms of variables that belong to other worker.

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 [7, 15] in AND_parallelism or by the user in OR_parallelism. The execution model has been developed as an extension of the Warren Abstract Machine (WAM) [18], maintaining both the optimization of the sequential Prolog techniques and the speedup of the Independent AND_parallel system and the multisequential OR_parallel system.

In the case of AND_parallelism, parallel goals are sent along with their variable bindings, to the idle worker indicated by the controller and an answer is then awaited by the parent worker. In order to control the execution of a parallel call, we have modified the shared memory model developed by Hermenegildo [12] for a distributed system.

The exploitation of OR_parallelism is based on the multisequential execution of the branches of the search tree, splitting the work dynamically. When a worker finds a parallel clause, it makes the new work available for idle workers, by sending a warning to the controller. Since each processor works in its own environment, the parent worker environment has to be reconstructed by the worker which assumes the new work. Instead of copying the environment as MUSE [1], PDP *recomputes* [2] the initial goal without backtracking, following the *success path* obtained from the parent worker (Figure 1). The amount of data communicated with this approach is smaller than that of the copying approach, which in turn speeds up the execution on a distributed system.

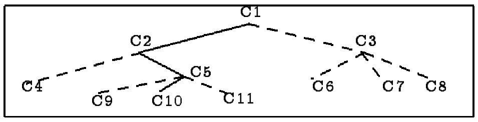

Figure 1: Recomputation following the success path C2,C5,C10

For programs presenting OR_under_AND parallelism the different solutions of each goal in a *parallel call* (set of parallel goals) have to be combined.

In the AND_parallelism approach the task carrying out a parallel goal reaches the successive solutions sequentially, when they are required because of backtracking. Therefore, if there is OR_parallelism in a parallel goal, maintaining the AND_parallelism approach requires the storing of the different solutions until required for building a new combination. The PDP approach avoids storing partial solutions and synchronizing workers. The idea is to create a new computation for each combination of solutions, recomputing the success path from the initial goal of the program until the parallel call is reached. In this way, the exploitation approach of AND_parallelism becomes that of OR_parallelism, which creates independent computations. This avoids the communications overhead of the AND_parallelism approach which is due to both the sending of solutions to the parent task and the requirements of new solutions from the offspring tasks. This is the reason why the speedup achieved by exploiting both kinds of parallelism may be greater than the product of the speedup achieved by exploiting each kind of parallelism separately. To avoid repeating solutions, we have introduced a *combination rule* to set the combinations corresponding to each new task.

The rest of the paper proceeds as follows: section 2 presents the execution model. Section 3 describes the parallel tasks of PDP. Section 4 presents a PDP execution scheme. Section 5 describes the scheduling policy. Results are presented in section 6 and finally conclusions drawn in section 7.

2 Execution Model

PDP exploits pure AND_parallelism, pure OR_parallelism and the combination of both, producing *OR_tasks* which explore the search tree branches from the root, and *AND_tasks* which execute parallel 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. Therefore in the combined model both kinds of parallelism are joined in a very natural way. Another important point is that the search tree is automatically distributed among the workers by means of a distribution rule, so that no worker is in charge of the distribution. The model is outlined as follows:

- The execution of a program begins as an OR_task, that records the success path.
- If AND or OR_parallelism appear, new AND or OR_tasks are created, respectively.
- 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 at this point has received this information for this purpose only. In this way, **the**

exploitation approach of AND_parallelism becomes that of OR_parallelism, and this reduces the exchange of information.

- Once the recomputation of this OR_task arrives to the parallel call from where it originates, if the operation were to blindly exploit the OR_parallelism, the result would be the simple repetition of solutions. In order to avoid this we have introduced a *combination rule* which decides the branch to be explored to solve each parallel goal. Let us call any goal which presents OR_parallelism and causes the creation of OR_tasks, the ancestor goal of these tasks. The key point of this rule is to fix the solution of the goals on the left of the ancestor goal and to combine them with every solution of the remaining goals. The *combination rule* is as follows:

  – If the goal is on the left of the ancestor goal, the branch to be explored is *fixed* to that explored by the previous "forefather" OR_task.

  – If the goal is the ancestor goal, the branch to be explored is the next one to that explored by the AND_task.

  – If the goal is on the right of the ancestor goal, the branch to be explored is the one leading to the first solution.

A new structure is introduced to specify the untried clauses: the *cross product environment* (CPE), associated to each parallel call. It is composed of a pointer marking the beginning of the success path corresponding to each goal in a parallel call.

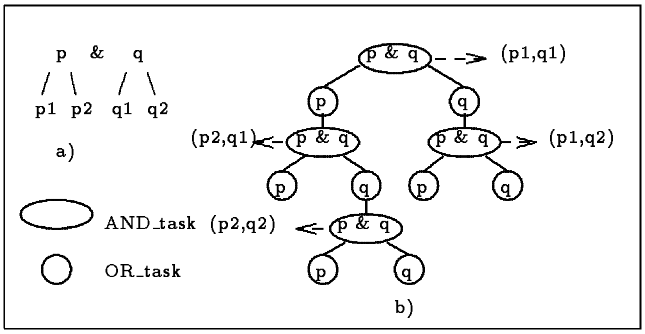

Figure 2: OR_under_AND parallelism in PDP

In order to visualize this process consider the program represented in Figure 2a), whose execution is shown in Figure 2b). When the AND_tasks which execute the parallel goals p and q find OR_parallelism, the first alternative clauses $p1$ and $q1$ are explored to give the answer to the parent task, and new OR_tasks are created to explore $p2$ and $q2$. For the OR_task corresponding to $(p2, q1)$ p is the ancestor goal and therefore, the branch to be explored so as to solve p corresponds to $p2$ ($p1$ is already being explored by the parent AND_task). Since q is on the right of the ancestor goal, the branch to be explored is $q1$, the first one. For the OR_task corresponding to $(p2, q2)$ the ancestor goal is q and therefore, the branch to be explored to solve p (which is on the left) is $p2$, the same as that explored by the previous forefather OR_task, i.e. that corresponding to $(p2, q1)$. For q the branch to be explored is $q2$, the following branch to the one explored by the parent AND_task.

3 Parallel Tasks in PDP

The PDP approach to exploit OR_under_AND parallelism leads to the distinction between different kinds of OR and AND_tasks depending both on the kind of task it arose from and on the ancestor goal position. The kinds of tasks are:

- **Primary OR_task:**
  This arises from the OR_parallelism exploitation in an OR_task. When the recomputation of the received success path is completed, the execution follows in the normal way.

- **Secondary OR_task:**
  This arises from the OR_parallelism exploitation in an AND_task. When the recomputation of the success path leading to the parallel call is finished, a new combination of solutions is created.

- **Primary AND_task:**
  This arises from the AND_parallelism exploitation in an AND_task, a primary OR_task or a secondary OR_task provided the latter does not correspond to a goal on the left of the ancestor goal. Primary AND_tasks exploit OR_parallelism appearing during the execution.

- **Secondary AND_task:**
  This arises from the AND_parallelism exploitation in a secondary OR_task corresponding to a goal on the left of the ancestor goal. According to the combination rule, this task must ignore any OR_parallelism appearing during the execution.

Producer and consumer relationships of the tasks can be shown pictorially as a *task tree*. In general, the task tree and the search tree are different, since it is not always possible, neither suitable to exploit all the potential parallelism of the program. Figure 3 shows a PDP execution example corresponding to the following program.

```
:- p & q.
p :- p1.          q :- q1.
p :- p2.          q :- q2.
p :- p3.
```

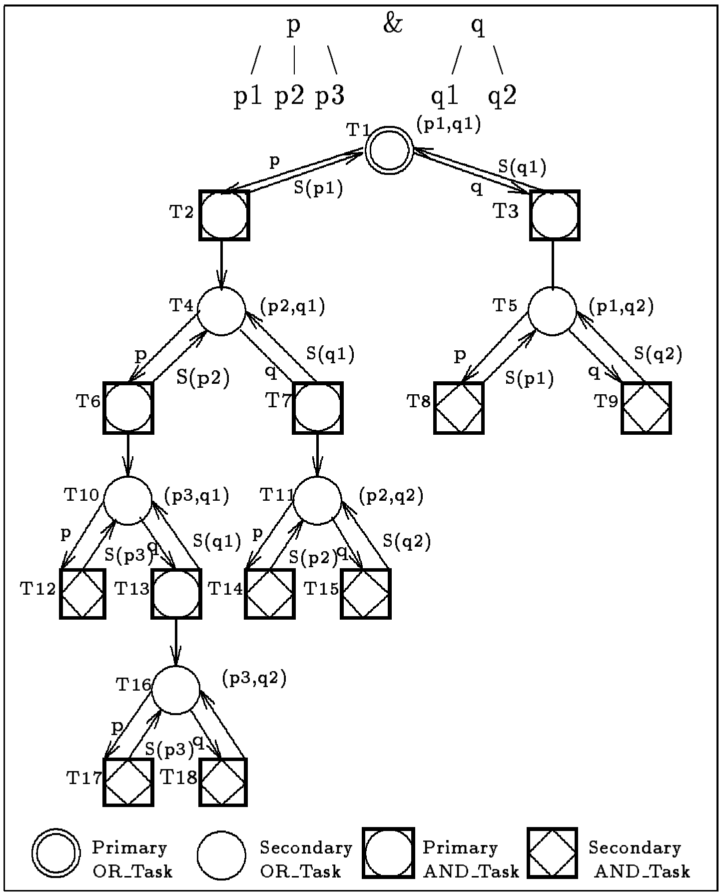

Figure 3: PDP execution example

The first solution of the parallel call $p\&q$ is obtained with the clauses $p1$ and $q1$ of p and q respectively. If a failure occurs or new solutions are required, there is a number of pending alternatives to obtain the solutions, corresponding to the cross product of p and q: $(p1, q2)$, $(p1, q3)$, $(p2, q1)$, etc. The execution begins as a primary OR_task that finds a parallel call and creates the AND_tasks T2 and T3. When the goal p is executed, T2 finds OR_parallelism, then it takes the first clause $p1$, explores it by itself and creates a new secondary OR_task, T4, for exploring a new solution. T1 builds the CPE $(p1, q1)$, that is passed to T2 and T3. T2 builds the CPE $(p1*, q1)$ because the ancestor goal (*) is the first one. The next combination corresponding to this CPE is passed to T4 which takes the next clause, $p2$, for the goal marked as the ancestor one, and the first one for every goal on the right, $q1$. T5 receives $(p1, q2*)$, indicating that the alternative clause to exploit for p is the first one and for q it is the second one.

```
type res_type = (success, fail);
type task_type = (primary_AND, secondary_AND, primary_OR, secondary_OR);
process AND_task(℘:program; Q:goal; V:substitution;
            C:success path; CPE: cross product env.; tasktype: task_type);
var
    R: resolvent; result: res_type; S: backtracking stack;
begin
    R := [Q]; S := ∅; C := C[CPE[Q]];
    recomputation(℘, R, S, V, C, CPE);
     if R <> ∅ then
        execution(℘,R,S,V,C,CPE,tasktype,result); end
    else result := success; end
    if result = success then send_success(parent_task, restriction(V,Q), alternative);
    else send_fail(parent_task);
end
```

Figure 4: AND_task creation

The scheme of creation of an AND_task is shown in Figure 4. The entry to an AND_task consists of a goal along with its variable bindings V, the success path C and the CPE corresponding to the received goal. The program and the kind of AND_task to be created are also received. The *resolvent* R and the *backtracking stack* S (where the state is saved) are initialized. The new AND_task takes from the received success path C the part which corresponds to the goal to be executed, starting at the point indicated in the CPE. The success path of the goal may be empty, incomplete or complete, depending on both the kind of task and on the position of the goal in the parallel call. If after the recomputation of the success path of the goal the resolvent is not empty, the pending execution is performed. The answer obtained is sent to the parent task.

```
process OR_task(℘:program;Q:goal; C:success path;
                        CPE: cross product env.; tasktype: task_type)
var
    S : backtracking stack; V: substitution; R: resolvent;
begin
    R := [Q]; S := ∅;
    recomputation(℘,R,S,V,C,CPE);
    execution(℘,R,S,V,C,CPE,tasktype,result);
    if result = success then
        V := restriction(V,Q);
    display(V,result);
end
```

Figure 5: OR_task creation

The scheme of creation of an OR_task is shown in Figure 5. An OR_task receives as entry the success path and the CPE of the parent task. The program along with the goal to be executed and the kind of OR_task to be created are also received. After initializing the resolvent and the backtracking stack, the recomputation of the success path is performed. This recomputation produces an environment consisting of a resolvent, a backtracking stack and a substitution. Then, the execution of the pending resolvent is performed. Once it is finished the result is presented to the user.

4 PDP Execution Scheme

Figure 6 shows the PDP execution scheme. Following the resolution algorithm, the resolvent is transformed until it becomes empty (successful execution) or a failure occurs. In each resolution step, a set of independent goals is taken from the resolvent (AND_parallelism). Each independent goal is solved by a different *AND_task* except the first one which is executed by the parent task itself. If the created AND_task is *primary*, the CPE is updated to a new combination, and the answers corresponding to goals executed by other tasks are awaited (*waited_answer* = 0). For the goals whose computation succeeds, the computed answer substitution obtained is received and composed with the computed answer substitution of the current task. If all computations succeed, the resolution process continues. The resolution of an only goal ($r = 1$) *unifies* the goal with the head of a clause of the associated predicate. If there is OR_parallelism, *OR_tasks* are created to perform the unification of the goal with other clauses, removing these pending alternative clauses in the parent processor. If the created OR_task is *secondary* the CPE is updated to a new combination. If the execution of the goal succeeds, the state is *saved* in the backtracking stack, the substitution obtained and the previous one are *composed*, the goal is *replaced* by the body of the clause in the resolvent, and the alternative clause considered is recorded in the *success path.* If the execution fails, backtracking will be performed taking a previous state out the backtracking stack.

5 Scheduling Policy

The controllers are responsible for the distribution of pending work among idle workers. At initialization time, each worker is loaded with the same program and the execution starts in one of them. A worker will be in one of the three states: *idle* (without work), *busy* (working) or *offering* (with pending work). The controller is warned about idle and offering workers informing to the idle workers which offering worker has to be requested for work. To optimize the communications, the workers do not report every change in

```
procedure execution(℘:program; R:resolvent; S:backtracking stack; V:substitution
    C: success path; CPE: cross product env.; tasktype:task_type; result:res_type);
var
    a: goal; P_a: associated procedure to goal a;
    c: clause; V': substitution; r: integer;
begin
  repeat
      [A_1, ..., A_r] := independent_set(R); r = size([A_1, ..., A_r]);
      if r > 1 then begin
          waited_answers := 0;
          for i := 2 to r  /* AND PARALLELISM */
              if tasktype = primary_OR_task or
              tasktype = primary_AND_task or
              tasktype = secondary_OR_task and ancestor_goal(CPE) <= i then
                  AND_task(℘,A_i,restriction(V,(A_i)),C,new_comb(CPE),primary);
              else
                  AND_task(℘,A_i,restriction(V,(A_i)),C,CPE, secondary);
              waited_answers := waited_answers + 1; end
          execution(℘, [A_1], S, V, C, CPE, tasktype, result);
          while waited_answers > 0 and result = success do begin
              result := receive_answer(i,θ_i);
              waited_answers := waited_answers - 1;
              if result = success then begin
                  R := apply(θ_i,R);
                  V := V ∘ θ_i; end
          end
      end
      else begin /* r = 1 */
          a := R[1] /* The first atom in the resolvent is taken */
          search_procedure(a,℘,P_a); /* associated procedure to the goal a */
        repeat
              c := P_a[1]; /* The first pending clause in the procedure is taken */
              P_a := P_a - c;
              if OR_parallelism(c) then
                  if tasktype = AND_task or tasktype = secondary_OR_task then
                      OR_task(℘,R,V,C,new_comb(CPE),secondary);
                  else
                      OR_task(℘,R,V,C,CPE,primary);
                  P_a := P_a - P_a[1]; end
              result := unify(a,c,V');
          until (P_a = []) or result = success;
          if result = success then begin
              if P_a <> [ ] then save(P_a,R,V,S);
              V := compose(V,V');
              replace(R,a,c); /* In R, replace a by c body */
              push(c,C); end /* The explored alternative is write in C */
      end
      if result = fail then backtracking(℘,S,R,V,C,result);
  until (R = [ ]) or result = fail;
end
```

Figure 6: PDP execution scheme

program	strategy A	strategy B	strategy C
query	3.4	3.2	3.3
zebra	3.9	3.5	3.6
mm	4.5	4.3	4.3
queen(8)	12.9	11.9	12.6
queen(9)	14.2	13.5	13.8
queen(10)	14.7	14.5	14.5

Table 1: Speed-up for different scheduling policies

their work load. The controller has exact information about idle workers and offering workers, and approximate information about the workers load.

In order to choose the offering worker which is going to share work with an idle worker, several strategies previously used by different systems [16, 13] have been tested.

- strategy A: Choose the nearest (physical distance) idle worker to the offering worker
- strategy B: Choose the most loaded offering worker
- strategy C: Choose the oldest offer

Table 1 shows the speedup on a system with 15 workers for each strategy. Since the measured times differ from run to run, all speedups given are computed using the average times of three runs. The example programs examined are standard benchmarks for AND_parallel systems, the *queen* problem, *query*, a database problem, *zebra*, a puzzle and *mm*, the mastermind program. The results show that the best strategy is A, choosing the nearest idle worker to the offering worker. This strategy optimizes the traffic in the network and favours exchanges between workers which have shared work previously, optimizing OR_parallelism exploitation. The worst strategy is B, since it is the most expensive one (it requires more information exchange than the other), while strategy C, choosing the oldest request is almost as good as A, since it is the cheapest one in that it doesn't need any analysis by the controller.

Because of the overhead associated with the parallel execution, the *granularity* of a job, i.e. an estimate of the amount of work needed to solve it, should be taken into account [8, 9, 17] when deciding whether or not to execute a job as a separate task. PDP applies control mechanisms based on heuristic observations about memory occupation.

Measurements have shown the similarity of the amount of data in the stack when each solution is reached. Therefore, it is possible to estimate how

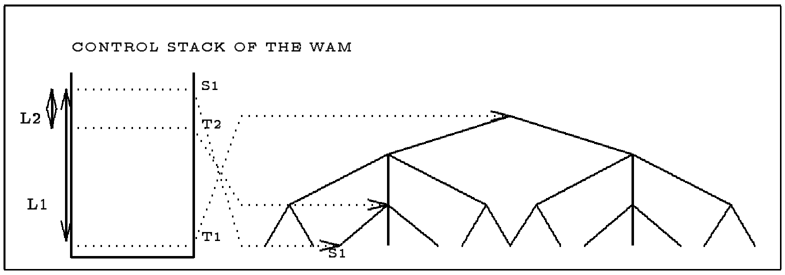

Figure 7: Granularity estimation of OR_parallelism exploitation

close a worker is to reaching the next solution and therefore if it is worthwhile to share the work that leads to the next solution. PDP performs a granularity control for the exploitation of OR_parallelism based on this observation. When a worker obtains a solution, it records the value of the backtracking stack top. The pending OR_tasks are associated to a *choice point* in the backtracking stack. A pending OR_task is sent to an idle worker only if the distance between the choice point and the top of the stack when the last solution was reached is greater then a critical value that is experimentally set. This test does not introduce run time overhead since only a comparison is needed. Figure 7 shows the stack size when solution $S1$ is reached. This point is very close to the choice point associated to the pending task $T2$ and therefore the distance $L2$ is smaller than the critical value C. This means, as the search tree shows, that $T2$ has a fine grain and therefore the task is kept in the worker. On the other hand, the task $T1$, that corresponds to a choice point in the bottom of the stack, has a high granularity and is sent to an idle worker.

	3 proc.		8 proc.		15 proc.	
program	no c.	with c.	no c.	with c.	no c.	with c.
query	2.6	2.6	2.8	2.9	3.0	3.4
zebra	1.8	1.8	3.5	3.7	3.6	3.9
mm	2.1	2.1	3.2	3.4	4.1	4.5
queen(8)	2.9	2.8	7.0	7.3	12.3	12.9
queen(9)	2.4	2.4	7.5	7.7	13.8	14.2
queen(10)	3.0	3.1	7.8	7.9	14.5	14.7

Table 2: Speed-up achieved controlling OR_parallelism granularity

Measurements have been taken to investigate the critical value C. On a system with 8 and 15 workers, this value, that depends on the system size is about $stack\_size/6$, where $stack\_size$ is the size of the backtracking stack when the last solution was reached. The speedup achieved by performing this procedure is shown in Table 2. For 3 workers the granularity control has no effect. For 8 workers the speedup increases for programs with a lot of parallel work (queen10). The greatest effect of the control is achieved on the system with 15 workers.

6 Performance Results

Our current implementation has been made in Parallel ANSI C on a Supernode (Parsys) with 16 T800 transputers connected in a torus network. In each processor the computation and communication functions have been split. There are three processes controlling the input, output and computation respectively.

The system does not support *cut* and database predicates yet, so the programs used to evaluate the system does not present these predicates. Some sequential programs have been tried in order to evaluate the overhead due to the parallel mechanism. The experiments demonstrate that the overhead when sequential programs are run on PDP is less than 5%. Results show that the overhead due to the writing down of the success path is less than 5%.

program	3 workers		8 workers		15 workers	
program	copying	recomp.	copying	recomp.	copying	recomp.
query	2.8	2.6	3.0	2.9	3.4	3.4
zebra	1.9	1.8	3.6	3.7	3.1	3.9
mm	2.2	2.1	3.4	3.4	4.5	4.5
queen(8)	3.1	2.8	7.4	7.3	12.9	12.9
queen(9)	2.4	2.4	7.7	7.7	14.1	14.2
queen(10)	3.0	3.1	8.0	7.9	14.0	14.7

Table 3: OR_parallelism speed-up with copying and recomputation approaches

We have compared speedup obtained exploiting OR_parallelism using stack copying and recomputation approaches. All benchmarks have been executed in a "program, fail" way. The results obtained are shown in table 3. These results show significant speedup for all tested programs. Though the achieved speedup using stack copying and recomputation is quite similar, it is slightly greater with the copying approach using a smaller number of workers, and for programs with smaller granularity. On the other hand, it becomes greater with the recomputation approach when the system size increases. The reason for this is the smaller amount of information exchange in the recomputation approach. For queen10 the copying approach is more efficient on a 3 or 8-worker system, while on a 15-worker system more efficiency is gained using the recomputation approach.

program	3 workers	8 workers	15 workers
qsort(700)	2.3	2.8	3.4
merge(500)	2.3	2.5	2.7

Table 4: AND_parallelism speed-up

The exploitation of AND_parallelism in PDP requires high granularity programs, since the workers sharing a job, exchange more messages than in the case of OR_parallelism. Table 4 shows the speedup for *mergesort* and *qsortsort* programs, standard benchmarks used for AND_parallel systems. For programs with more fine grain parallelism no speedup is achieved by exploiting AND_parallelism.

In order to evaluate the behavior of PDP for programs with both kinds of parallelism, synthetic benchmarks with coarse grain parallelism have been run. The first one (synthetic 1) presents AND_under_OR parallelism:

```
:- check.
check :- times1(X),(p(X) & p(X) & p(X)).
check :- times2(X),(p(X) & p(X) & p(X)).
check :- times3(X),(p(X) & p(X) & p(X)).
times1(2000). times2(1000). times3(500).
p(0).
p(X) :- X > 0, X1 is X - 1, p(X1).
```

There is OR_parallelism in the procedure *check* while AND_parallelism appears in the body clauses of this procedure. The following benchmark (synthetic 2) presents OR_under_AND parallelism:

```
:- check(X).
check([Xs,Ys,Zs]) :- times(X), times(Y), times(Z), (p(X,Xs) & p(Y,Ys) & p(Z,Zs)).
p(X,Xs) :- p1(X,Xs).
p(X,Xs) :- p2(X,Xs).
p1(0,a).
p1(X,Xs) :- X > 0, X1 is X-1, p1(X1,Xs).
p2(0,b).
p2(X,Xs) :- X > 0, X1 is X-1, p2(X1,Xs).
times(1000).
```

There is AND_parallelism in the body of the *check* clause, while the procedure *p* presents OR_parallelism. Table 5 presents the speedup achieved by exploiting each kind of parallelism with 15 workers. Results show signifi-

program	OR_par.	AND_par.	Comb. par.
synthetic 1	1.5	2.9	4.5
synthetic 2	2.4	1.7	4.4

Table 5: OR, AND and Combining parallelism speed-up

cant speedup for all executions exploiting parallelism. The benchmark 1 has been chosen to show the advantages of exploring at the same time different solutions since the first solution explored by the sequential machine can be the slower to reach (the second clause of *check* is computed in a shorter time that the first one). It may be observed from the table that when both kinds of parallelism are exploited, the performance has been improved in all cases. The speedup achieved when exploiting both kinds of parallelism is greater than the product of the speedups achieved by exploiting each kind of parallelism separately. The reason is that the exploration of the different solutions when AND_parallelism is exploited requires a number of messages exchanges between the parent worker and the worker exploring each goal in the parallel call (new solution requests and answers) that are avoided when OR_under_AND parallelism is exploited.

7 Conclusions

A system, PDP, to execute Prolog programs on distributed memory systems exploiting both, Independent_AND and OR_parallelism, has been presented. The execution model is based on multisequential Prolog engines that work independently under a hierarchic control. PDP deals with OR_under_AND parallelism producing in a distributed way the cross product of the solutions of the goals in a parallel call and creating a new computation for each combination. This avoids both the storage of partial solutions and the synchronization of workers. Results show that the overhead introduced to sequential execution is quite small. The overhead due to the exploitation mechanism of each kind of parallelism is also small. Different scheduling policies have been tested. The best result have been achieved when the nearest idle worker was chosen. This strategy is expected to yield even better results when the system size increases. Granularity controls have been introduced, showing a performance improvement when the system size increases. Significant speedup is achieved for coarse grain benchmark programs with each kind of parallelism. For some programs presenting both kinds of parallelism PDP achieves better results than the product of both. In the future the system will be extended to a larger number of workers and real applications will be tried.

Acknowledgements

We would like to thank Mario Rodriguez Artalejo and Jose Cuesta for their helpful comments.

References

[1] Ali, K. A. M, Karlsson, R.. *The Muse Approach to Or-Parallel Prolog.* Int. J. of Parallel Programming, Vol 19 No. 2 (1990), pp. 129-162.

[2] Araujo, L. and 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.

[3] Biswas P., Su S., Yun D. *A Scalable Abstract Machine Model to Support Limited-OR(LOR)/Restricted-AND Parallelism(RAP) in Logic Programs.* Proc. ICLP (1988), pp. 1160-1179.

[4] Calderwood A., Szeredi, P. *Scheduling Or-parallelism in Aurora - the Manchester scheduler* Proc. ICLP (1989), pp. 419-435.

[5] Clocksin W. F., Alshawi H. *A Method for Efficiently Horn Clause Programs Using Multiple Processors* New Generation Computing,5 (1988), pp. 361-376.

[6] Conery, J. S. *Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors* Int. J. of Parallel Programming 17(2), (1988), pp. 125-152.

[7] Chang, J. -H., Despain, A. and Degroot, D. *AND-parallelism of logic programs based on a static dependency analisys*, Proc. Spring Compcon, IEEE, (1985), pp. 218-225.

[8] Debray, S.K., Lin, N.-W., Hermenegildo, H. *Task Granularity Analysis.* SIGPLAN'90 Conf. on Programming Language Design and Implementation, (1990), pp. 174-188.

[9] Debray, S.K., Lin, N. *Automatic complexity Analysis of Logic Programs* Proc. ICLP. (1991), pp. 599-613.

[10] Degroot, D., *Restricted AND-Parallelism.* Proc. of Int. Conf. Fifth Gen. Comp. Sys., ICOT (1984), pp. 471-478

[11] Gupta, G., Hermenegildo, M. *ACE:And/Or-parallel Copying-based Execution of Logic Programs.* Tech. Report TR-91-25, Department of Computer Science, University of Bristol, Oct 1991.

[12] 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).

[13] Kuchen, H., Wagener, A. *Comparison of Dynamic Load Balancing Strategies* Tech. Report 90-5 . Aachener Informatik-Berichte.

[14] Lin, Y.-J. and Kumar, V., *AND-parallel execution of Logic Programs on a Shared-Memory Multiprocessor.* The J. of Logic Programming, (1991):10:, pp. 155-178.

[15] Muthukumar, K., Hermenegildo, M. *Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation* North American Conf. LP (1989) pp. 166-185.

[16] Sugie, M. Yoneyama, M., Tarui, T. *Load-Dispatching Strategy on Parallel Inference Machine* Proc. Int. Conf. on Fifth Generation Computer System (1988), pp. 987-993.

[17] Tick, E. *Compile-Time Granularity Analysis for Parallel Logic Programming Languages* New Generation Computing, 7 (1990), pp. 325-337.

[18] Warren, D.H.D., *An Abstract Prolog Instruction Set.* Tech. Note 309, SRI International, (1983).

[19] Westphal, H, Robert, P, Chassin, J, Syre, J. *The PEPsys model: Combining Backtracking, AND- and OR-parallelism.* Proc. Symp. of LP (1987), pp. 436-448.