A PARALLEL PROLOG SYSTEM FOR DISTRIBUTED MEMORY

LOURDES ARAUJO AND JOSE J. RUZ

▷ This paper presents a parallel execution system (PDP: Prolog Distributed Processor) for efficiently supporting both Independent_AND\OR parallelism on distributed memory multiprocessors. The system is composed of a set of workers with a hierarchical structure scheduler. Each worker operates on its own private memory and interprocessor communication is performed only by the passing of messages. The execution model follows a multisequential approach in order to maintain the sequential optimizations. Independent AND_parallelism is exploited following a *fork-join* approach and OR_parallelism is exploited following a *recomputation* approach. PDP deals with OR_under_AND parallelism by producing the solutions of a set of parallel goals in a distributed way, that is, by 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. Different scheduling policies have been studied, and granularity controls have been introduced for each kind of parallelism. PDP has been implemented on a network of transputers 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

The declarative semantics of logic programming languages in general [22, 23], and of Prolog in particular, allows programmers to work at a high level of abstraction and

Supported by the Prontic project TIC92-0793-C02-01.
This paper is an extended version of [5].
Address correspondence to Dpto.Informática y Automática, Universidad Complutense de Madrid, Madrid 28040, Spain
{lurdes, jjruz}@dia.ucm.es

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

enable system designers to exploit implicit parallelism in the programs, thus improving the performance. Prolog programs present inherent parallelism that can be exploited by evaluating the multiple solutions of a goal in parallel (OR_parallelism), or by simultaneously executing the goals in the body of a clause (AND_parallelism).

The goals in the body of a clause can share variables, and if so, the parallel execution of these goals must be coordinated in order to avoid the same variable to be bound to different values. One approach to avoid this problem is to identify independent goals by using a static analysis of the program at compile-time. However, this approach may miss many opportunities of parallelism. Conery [11] proposed a complex system that dynamically creates a dataflow graph in which a generator-consumer relationship among goals is established. Goals are solved in the order specified by the graph. This approach provides a great amount of parallelism; however, it can involve a large overhead. Degroot [15] has developed a method which combines compile-time analysis with run-time checking. The compilation of a program creates a Conditional Graph Expression (CGE) for each program clause and expresses potential AND_parallelism. Then, the run-time overhead of detecting binding conflicts is substantially reduced by only having to check the conditions of the CGEs. Hermenegildo [19] proposed a WAM-based implementation for AND_parallelism exploitation in shared memory systems that incorporates the Degroot approach including backward computation.

In OR_parallelism, if the implementation assumes a common memory space for the execution of different branches of the AND\OR tree, it is necessary to represent different bindings of the same variable corresponding to different branches of the search tree. A number of models have been proposed to deal with this issue. For instance, the SRI model [33] keeps the multiple variable binding in a binding array, and has been adopted as the storage model for the Aurora system [9]. Another instance is the Argonne model [34], which uses a hash array. Systems such as MUSE [2] and Delphi [3], with a distributed memory space for each parallel execution, do not need a representation for the multiple bindings of a variable. The MUSE system transfers an explicit copy of the data of a process which is performing an execution, to the process that is going to collaborate in the execution. The Delphi model reconstructs a process environment by recomputing the initial goal, controlling the alternative search paths by a set of bit strings, called oracles.

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. A considerable number of approaches have been recently proposed for AND-OR parallel execution [34, 7, 16, 21, 17], most of which are implemented on systems with total or partial shared memory. While shared memory systems have the advantage of avoiding communication overhead, systems with distributed memory have other attractive features, such as their scalability or their more widespread use (transputers, clusters of workstations, etc). Because of these reasons it is interesting to study how to implement parallelism on a distributed memory system.

In this paper we present a model we call Prolog Distributed Processor (PDP), a multisequential system supporting both Independent_AND and OR_parallelism as well as the combination of both. The development of the system so far corresponds to the pure language (horn clauses). Some ideas to extend the system with side-effect predicates are pointed in section 9.

PDP is composed of a pool of processors organized as a set of clusters, each consisting of a *scheduler* and a set of *workers*. Schedulers 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. A higher level scheduler will take care of the distribution of pending work among different clusters. Schedulers are exclusively communicated by message passing. In this way, the scheduling policy has a hierarchical structure, and the number of levels depends on the number of workers in the system. 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* [12], that is, there are no variables in a worker defined in terms of variables that belong to another worker. The execution model has been developed as an extension of the Warren Abstract Machine (WAM) [32], which has been recognized for years as the fastest implementation of Prolog. It is an efficient execution model and compilation technique with several optimizations that notably increase its performance. In our extension we have taken care of retaining the WAM memory-management efficiency as well as its performance optimizations. On the other hand, we have also maintained the speedup achieved by the exploitation of each kind of parallelism when they appear separately.

Independent AND_parallelism is exploited by following a *fork-join* scheme: parallel goals are sent along with their variable bindings (thus in the form of a closed environment [12]) to the idle worker indicated by the scheduller, and an answer is then awaited by the parent worker. In order to control the execution of a parallel call, we have adopted an extension for distributed memory systems of the RAP model for shared memory systems by Hermenegildo [19].

The exploitation of OR_parallelism is based on the multisequential execution of the branches of the search tree, splitting the work dynamically. A worker that finds a parallel clause makes the new work available for idle workers. The parent worker environment is reconstructed by using a recomputation approach (a technique first introduced by the Delphi group [3]). In PDP the recomputation of the goal avoids backtracking by following the socalled *success path*, i.e. the sequence of clauses that have succeeded, obtained from the parent worker (Figure 1.1). (For a detailed account of the OR_parallel execution model see Ref. [4].) Although recomputation is not significantly more advantageous than a copying approach as concerns OR_parallelism [4], recomputation allows exploiting OR_under_AND parallelism in a very natural way.

The PDP approach to exploit OR_under_AND parallelism [5] is designed to create, in an automatic and decentralized way, an independent computation for each solution. Appearance of OR_under_AND parallelism takes place when at least one of the goals of a certain parallel call can be solved with more than one clause (OR_parallelism). The PDP approach can be interpreted as an application of the distribution law of logic:

$$(p_1 \vee p_2) \wedge q = (p_1 \wedge q) \vee (p_2 \wedge q)\,.$$

The left-hand side is an OR_under_AND sentence, while the right-hand side is an AND_under_OR sentence. Now the OR_parallelism can be exploited as usual in PDP, i.e. by splitting off the component clauses into independent branches of computation (each with a pure AND_parallel call), thus taking advantage of the recomputation technique. In this way, the execution of OR_under_AND parallelism becomes that of AND_under_OR parallelism, and OR_parallelism is more suitable to be implemented on distributed memory systems. This scheme is applied recursively:

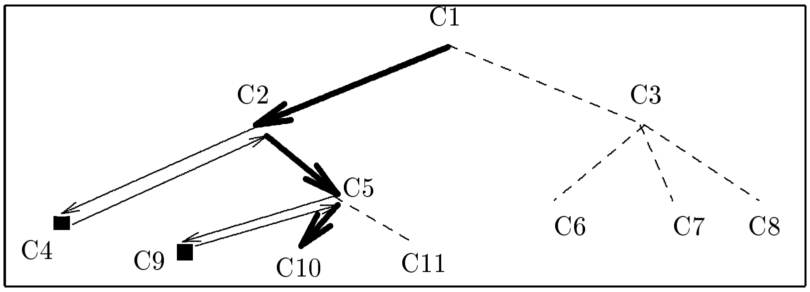

FIGURE 1.1. Example of success path: C2,C5,C10 is the success path since C4 and C9 have failed and backtracking has been performed, while the branches corresponding to C2,C5,C10 have succeeded so far.

when a computation has to execute a parallel call with more than one solution, it computes one of them and creates a new computation to obtain the remaining solutions. Of course an appropriate algorithm has to be devised in order to avoid repetitions. To this purpose, we have introduced a *combination rule* which sets, in a well defined manner, the solution corresponding to each new computation.

This approach has two obvious advantages: first, it avoids storing solutions because no parallel call is waiting for the completion of the whole set of solutions anymore, and second, the splitting off algorithm can resolve any OR_under_AND parallel call in an automatic, decentralized way, i.e. no processor has to perform the splitting, but anyone which gets idle selects, by applying the algorithm, its corresponding solution.

The rest of the paper proceeds as follows: section 2 describes the way of annotating a program for parallel evaluation; section 3 introduces the computational model; section 4 describes the execution model; section 5 presents the WAM extension for the implementation of PDP; section 6 describes the scheduling policy; section 7 presents granularity controls; section 8 illustrates the performance of the system, and finally, conclusions are drawn in section 9.

2. PARALLELISM ANNOTATION IN THE SOURCE PROGRAM

PDP exploits the parallelism that has been annotated in the source program. These annotations may have been carried out by a precompiler[10, 27, 8] or by the user himself. The system &-Prolog [20] provides an *annotator* for AND_parallelism which performs *dependency* analysis on the input code and also check any user-provided annotations. AND_parallelism is annotated by the & operator placed between the independent goals. This set of goals is then named a *parallel call.* For instance, $(B_k\&...\&B_j)$ is a parallel call appearing in the next clause:

$$A:-B_1,...,(B_k\&...\&B_j),...,B_n.$$

The parallel execution may depend on conditions on the groundness or independence of variables appearing in the parallel call. For instance, in the next clause, the parallel call $(B_k(X_k)\&...\&B_j(X_j))$ would be executed in parallel provided the variables $X_k, ..., X_j$ are all independent.

$$A : -B_1, ..., independent(X_k, ..., X_j) \Rightarrow (B_k(X_k)\&...\&B_j(X_j)), ..., B_n.$$

OR_parallelism allows the parallel execution of sets of clauses belonging to the same procedure (*parallel procedure*). The clauses in each set are then executed sequentially. OR_parallelism is annotated by placing the $*$ operator on the left of the first clause in each set. For instance, let us consider the next procedure. Annotating with $*$ sentences B :- B_1,... and B :- B_j,... means that sentences from B :- B_1,... to B :- B_{j-1},... form a subset and sentences from B :- B_j,... to the last one a second subset, and that both subsets can be executed in parallel.

```
*B :- B_1,...
 B :- B_2,...
       ...
       ...
*B :- B_j,...
 B :- B_{j+1},...
       ...
       ...
```

In general, a procedure annotated with OR_parallelism is expected to produce complex enough computations so as to deserve parallel execution. However, a user who knows the procedural behavior of the program may decide the sequential execution of a procedure annotated with OR_parallelism when it is invoked to solve a particular goal. Thus, the system provides a mechanism to disable the exploitation of parallelism in the execution of a particular goal. This is done by enclosing the goal between "{ }". The mechanism may also be applied to goals belonging to a parallel call. For instance, in the next clause, the goals B_k and B_j are executed exploiting only AND_parallelism, in spite of the fact that the procedures for $B_k, ..., B_j$ present OR_parallelism.

A :- $B_1, ..., (\{B_k\}\&...\&\{B_j\}), ..., B_n$.

```
*B_k :- ...
 B_k :- ...
       ...
       ...
*B_j :- ...
 B_j :- ...
```

Similarly, it is possible to disable the exploitation of AND_parallelism during the execution of a particular goal. This is done by enclosing the goal between "$<$ $>$". For instance, the goals B_k and B_j in the next clause are executed without exploiting the AND_parallelism that appears in the computation.

$$A \text{ :- } B_1, ..., (< B_k > \&...\& < B_j >), ..., B_n.$$

$$
\begin{array}{l}
{}^*B_k \text{ :- } ...(C1\&...\&Cm_c)... \\
\;\, B_k \text{ :- } ...(D1\&...\&Dm_d)... \\
\qquad ... \\
\qquad ... \\
{}^*B_j \text{ :- } ...(E1\&...\&Em_e)... \\
\;\, B_j \text{ :- } ...(F1\&...\&Fm_f)... \\
\qquad ... \\
\qquad ...
\end{array}
$$

Finally, it is possible to request the sequential execution of a particular goal by disabling the exploitation of both kinds of parallelism, as for the goals B_k and B_j in the next clause:

$$A \text{ :- } B_1, ..., (< \{B_k\} > \&...\& < \{B_j\} >), ..., B_n.$$

The use of the annotation to disable the parallelism exploitation is illustrated for the next program:

```
*p(X,Y,Xs,Ys) :- p1(X,Xs) & p2(Y,Ys).
 p(X,Y,Xs,Ys) :- p2(X,Xs) & p1(Y,Ys).

 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, p1(X2,Xs).
```

The goal ":- p(1000,500,X1,Y1), p(500,100,X2,Y2), p(100,20,X3,Y3)" has three independent goals that can be executed with AND_parallelism. The procedure p, that matches the goals, presents AND_parallelism and OR_parallelism. However, the size of the data of the second goal suggests that this goal is not complex enough for AND_parallelism exploitation. Similarly, the size of the data of the third goal suggests a sequential execution. Therefore, the goal should have the following form:

```
:- (p(1000,500,X1,Y1) & <p(500,100,X2,Y2)> & {<p(100,20,X3,Y3)>})
```

The three goals are executed in parallel, but OR_parallelism of p is exploited only in the execution of the two first goals, whereas the AND_parallelism of p is exploited only in the first goal.

3. COMPUTATIONAL MODEL

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 that 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* 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 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*. If the parent task has executed the goals $(g_1, ..., g_n)$ until the appearance of OR_parallelism, and the clause selected to solve each g_i has been C_{i,j_i} (where j_i labels the clause chosen in the procedure for g_i), then the new OR_task computes a new branch with clauses $C_{1,j_1}, ..., C_{n-1,j_{n-1}}, C_{n,j_n+1}$. In this way, the parent task provides this new OR_task with its computed answer substitution and the *list of choice points*, that is, the list of points in the search tree with pending clauses. An OR_task finishes when every assigned branch has been explored.

- 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, we have introduced a *combination rule* which decides which branch is explored to solve each independent goal. Let us first introduce some notation.

  The *solution S to a parallel call* $(g_1, ..., g_n)$ 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_ith 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.

  Then, 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:

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

- 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 $S = \{s_{1,j_1}, ..., s_{i-1,j_{i-1}}, s_{i,j_i}, s_{i+1,j_{i+1}}, ..., s_{n,j_n}\}$; if $j_i < m_i$, then S' will be:

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

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 of the goals on the left of the ancestor goal and to combine them with *every* solution of the remaining goals. It is straightforward to see from the definition of the rule that solutions do not repeat, since each solution and its previous one have a different local solution for the ancestor goal. A little more thought should also convince the reader that every solution is explored (notice that in the execution of the parallel call, every independent goal g_i plays the role of ancestor goal for a branch of the task tree).

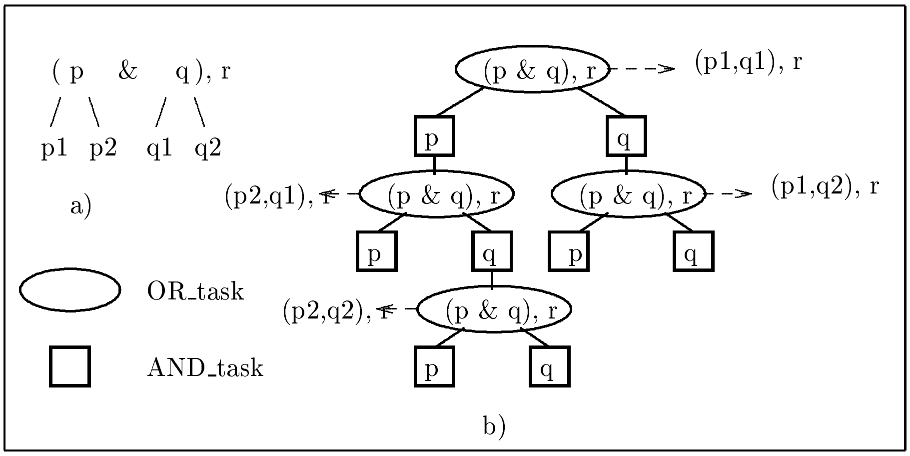

FIGURE 3.1. PDP execution scheme of a parallel call whose goals present OR_parallelism. The goal r is included to illustrate how PDP deals with a continuation goal.

In order to visualize how this combination rule works, consider the program represented in Figure 3.1a), whose execution is shown in Figure 3.1b). 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 so as to solve p (which is on the left) is $p2$, (the same as that explored by its predecessor 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. Finally, once a solution to each goal of the parallel call has been achieved, the remaining continuation goals (r) are executed. This has to be done for every combination of solutions so as to keep computations completely distributed.

4. EXECUTION MODEL

The execution model for the computational model just described, has been designed assuming a distributed memory system as support. The main points in the execution model are concerned with the execution of the different types of task, which correspond to the exploitation of each kind of parallelism, as well as the implementation of the combination rule. The model is summarized in the following points:

- *OR_tasks*
  The execution of a program begins as an OR_task that executes the initial goal and exploits the parallelism by creating new tasks. OR_tasks which appear later receive the answer substitution and choice point list of its predecessor OR_task; that is, a new OR_task needs the whole execution environment of its predecessor OR_task. A PDP OR_task obtains this environment by recomputing the initial goal without backtracking, following the *success path* of the predecessor OR_task. Therefore, an OR_task receives the success path of its predecessor OR_task, and recomputes the initial goal following this path. After the recomputation, the execution of the next solution is computed sequentially.

- *AND_tasks*
  A new AND_task receives a goal belonging to a parallel call and the answer substitution restricted to the variables of this goal. The AND_task executes the goal. The solution thus obtained is then given to the parent task along with the success path corresponding to the execution of the goal.

- *Implementation of the combination rule*
  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. The information needed to apply the rule is the success path of the predecessor OR_task (that is equivalent to the

solution explored for this task) and the ancestor goal position, and thus the solution to be explored is automatically known. A new structure, the *cross product environment* (CPE), is introduced in order to specify the beginning of the success path corresponding to each goal in a parallel call.

The next subsection gives a more precise presentation of each type of task and the resulting execution algorithm.

4.1. Parallel Tasks in PDP

The PDP approach to exploit OR_under_AND parallelism leads to a distinction between different types of OR and AND_tasks. The type of a task depends on both the type of its parent task and the ancestor goal position. The types of tasks are:

- **Primary OR_task:**
  This is created by an OR_task to exploit OR_parallelism. When the recomputation of the received success path is completed, the execution proceeds in the normal way.
- **Secondary OR_task:**
  This is created by an AND_task to exploit OR_parallelism. When the recomputation of the success path leading to the parallel call is completed, a new combination of solutions is created.
- **Primary AND_task:**
  This is created to exploit AND_parallelism by 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 is created to exploit AND_parallelism by 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.

The task tree may be composed of all these types of tasks. Figure 4.1 shows a PDP execution example corresponding to the following program.

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

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 are 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, to explore a new solution. T1 builds the CPE $(p1, q1)$, which is passed to T2 and T3. T2 builds the

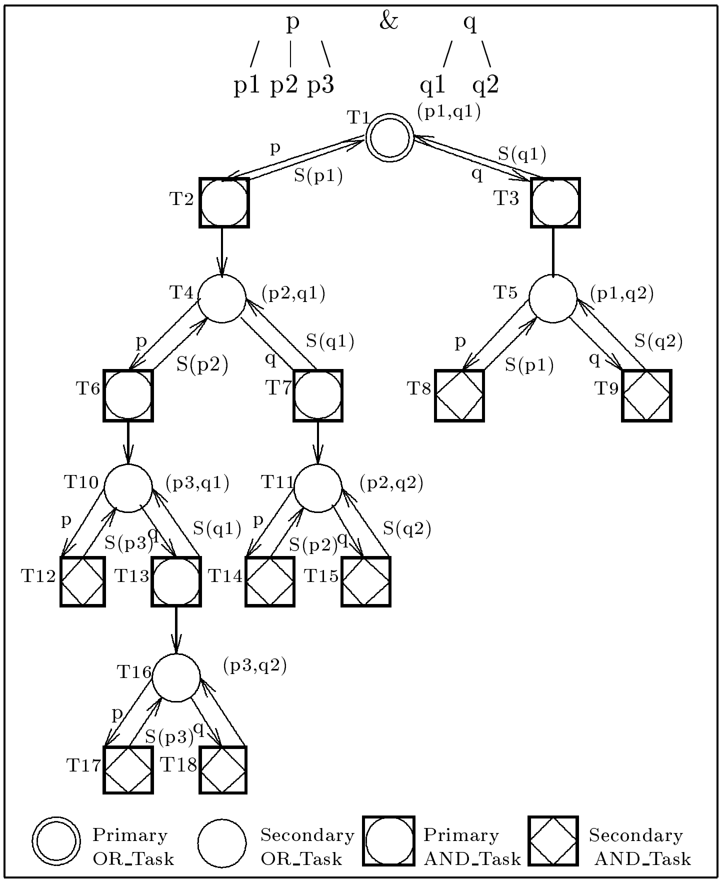

FIGURE 4.1. PDP execution example

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 clause 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 is the second.

The algorithm of an AND_task is shown in Figure 4.2. The input 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 type 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 type of task and 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 will be performed. The answer thus obtained is then sent to the parent task, including the success path corresponding to the goal

```
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; C': success path;
begin
    R := [Q]; S := ∅; C := C[CPE[Q]]; C' := ∅;
    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), C-C');
    else send_fail(parent_task);
end
```

FIGURE 4.2. AND_task

$(C - C')$.

```
process OR_task(℘:program; C:success path; CPE: cross product env.;
                tasktype: task_type)
var
    Q: goal; S: backtracking stack; V: substitution; R: resolvent;
    C': success path; result: res_type;
begin
    Q := first_goal(℘);
    R := [Q]; S := ∅;
    C' := next_path(C);
    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 4.3. OR_task

The algorithm of an OR_task is shown in Figure 4.3. An OR_task receives as input the success path and the CPE of the parent task. The program and the type of OR_task to be created are also received. After initializing the resolvent and the backtracking stack, the success path leading to a new solution is built (new_path(C)) by changing the last clause of the success path C by the next one in the procedure. Then, the recomputation of the new success path C' 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 this is completed, the result is presented to the user.

```
procedure execution(℘:program; R:resolvent; S:backtracking stack; V:substitution
    C: success path; CPE list: cross product env.; tasktype:task_type; result:res_type);
var
    a: goal;      Pa: associated procedure to goal a;
    c: clause;      V': substitution;      r, i: integer;
    θ: array[1..NMAX] of substitution; result: array[1..NMAX] of res_type;
begin
  repeat
    [A1, ..., Ar] := independent_set(R); r = size([A1, ..., Ar]);
    if r > 1 then begin
      forall i := 1 to r do/* AND PARALLELISM */
        if tasktype = primary_OR_task or tasktype = primary_AND_task or
        tasktype = secondary_OR_task and ancestor_goal(CPE) >= 1 then
          AND_task(℘,Ai,restriction(V,(Ai)),C,CPE,primary, result_i, θi);
        else
          AND_task(℘,Ai,restriction(V,(Ai)),C,CPE,secondary, result_i, θi);
      i := 1;
      while (i <= r) and result = success do begin
        recv_result(j,result_j,θj,Cj);
        result := result_j;
        if result = success then begin
          R := apply(θj,R); V := V ∘ θj;
          R := apply(θj,R);
          CPE[j] := end(Cj);
          C := C ∘ Cj;
          i := i + 1; end end
      CPE := CPE + r;
    end
    else begin /* r = 1 */
      a := R[1] /* The first atom in the resolvent is taken */
      search_procedure(a,℘,Pa); /* associated procedure to the goal a */
     repeat
        c := Pa[1]; /* The first pending clause in the procedure is taken */
        Pa := Pa - c;
        if OR_parallelism(c) and tasktype ≠ secondary_AND_task then
          if tasktype = AND_task or tasktype = secondary_OR_task then
            OR_task(℘,C,CPE,secondary);
          else
            OR_task(℘,C,CPE,primary);
          Pa := Pa - Pa[1]; end
        result := unify(a,c,V');
      until (Pa = []) or result = success;
      if result = success then begin
        if Pa <> [ ] then save(Pa,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 4.4. PDP execution algorithm

4.2. PDP Execution Scheme

Figure 4.4 shows the PDP execution algorithm. The execution environment consists of the *resolvent R*, the *backtracking stack S*, the *answer substitution V*, the *success path C* and the *cross product environment CPE*. Accordingly to the resolution algorithm, the execution consists of a loop of resolution steps which finishes either when the resolvent becomes empty (successful execution) or when a failure occurs. Two kinds of resolution steps are distinguished depending on the appearance of AND_parallelism. If *AND_parallelism* appears, a set of independent goals $[A_1, ..., A_r]$ is taken from the resolvent and a new AND_task is created to execute each one of them. The type of the new AND_tasks depends on the type of the parent task and on the ancestor goal position, as explained in 4.1. Each of these tasks receives the program, the assigned goal A_i, the binding of the goal variables (*restriction(V,*A_i*)*), the success path and the CPE indicating the part of the success path which corresponds to each goal. The execution of these AND_tasks produces a result (*result*$_i$), that may be *success* or *failure*. If the result is success, the AND_task provides the computed answer substitution (θ_i) corresponding to the goal, which is applied to the pending resolvent ($apply(\theta_i, R)$) and composed with the general substitution ($V := V \circ \theta_i$). The success path corresponding to the goal is also received and it is composed with the previous one, recording its position in the CPE. In this way, the CPE indicates the solution computed, and so, a new task is able to determine from the received CPE the solution it has to compute.

The other kind of resolution step corresponds to the execution of a single goal. If the clauses of the procedure associated to the goal present *OR_parallelism*, this is exploited by creating OR_tasks, whose type depends on the type of the parent task (see 4.1). The alternatives given to the created OR_tasks are erased ($P_a := P_a - P_a[1]$) in the parent task. If the unification of the goal with some of the remaining clauses in the task succeeds, the state is stored in the backtracking stack, the substitution is composed with the general substitution, the unified goal is replaced in the resolvent by the goals in the body of the selected clause, and the clause is recorded in the success path. If a resolution step produces a failure, *backtracking* is performed and a previous state is restored from the backtracking stack.

5. WAM EXTENDED FOR PDP

In order to reduce the communications overhead, PDP has been designed with a hierarchical scheduling policy. PDP is composed of a set of clusters, each of them consisting of a *scheduler* and a set of *workers*. Schedulers are responsible for the distribution of pending work among idle workers. Each worker operates on its own private memory and interprocessor communication is performed only by the passing of messages. A worker executes a task of any type until it is finished, then executes a new one, and so on. Each worker implements an extension of the Warren Abstract Machine (WAM) [32] which consists of the addition of new data structures and instructions to manage parallelism; more precisely, it is devised so as to deal with the following requirements:

- *Recording of the success path and the CPE list*
  The workers record the clause succeeding in each procedure call, and update

the success path when backtracking occurs. They also record the alternative clauses tried for each goal of a parallel call.

- *Recomputation*
  The workers which have received the task of exploiting OR_parallelism are able to follow the path they receive. To explore the parallel call appearing during the recomputation, the received CPE list is checked. The recomputation is optimized by taking advantage of the common information between successive tasks a worker is assigned, since only the different part need to be recomputed. We reduce the recomputation time by comparing the received success path and the previous success path and the skipping the recomputation of the common part.

- *Fork and join control of a parallel call*
  The parallel execution of a parallel call requires the synchronization of the reception of the answers.

- *Behavior depending on the type of assigned task*
  The type of the task determines different modes of working. In a secondary AND_task, OR_parallelism is not exploited. In a secondary OR_task, after completing the recomputation, a new combination of solutions is computed. A secondary OR_task begins in a special mode, GUIDED, in which the task performs an execution following the success path of the parent task. Once the recomputation is completed, the task operates in FREE mode, that is, creating choice points and recording its own success path.

The implementation of these extensions introduces little overhead when the execution is sequential or when pure parallelism (either AND or OR) is exploited. In what follows we describe in detail the extensions of the WAM we have introduced.

5.1. New Data Structures

The data structures added to record the *success path* are the *success stack* and the *success pointer* (SP). A new field has also been added to the choice points, containing the success register value, in order to automatically retrieve the success stack during backtracking. The selection of the recomputation mode is implemented by means of a new switch GUIDED/FREE. A *pending alternatives table* and a *counter* have also been added for the exploitation of OR_parallelism.

The *Cross Product Environment* (CPE) associated with each parallel call has been introduced in order to perform the combination of solution explained in section 3. This structure is composed of the ancestor goal of the parallel call, the number of goals, and a pointer for each goal which points to its corresponding path in the success path. These pointers make it possible to order the path corresponding to the goals in a parallel call which in general are received in an arbitrary order. The CPE's are stored in a new stack, whose top is pointed to by the *Cross Product Register* (CPR).

In order to synchronize the execution of a parallel call, we have adopted an extension for distributed memory systems of the system designed by Hermenegildo [19] for shared memory systems. These structures, stored in the Stack, are the *Parallel Call Environment* (PCE), which records the evolution of the independent goals execution; the *Local Goal Marker* (LGM), which is used by an OR_task to identify a local goal as belonging to a parallel call; the *Remote Goal Marker* (RGM), which is used by an AND_task that executes a goal of a parallel call, in order to store information about its parent task, and finally, the *Parallel Call Completion*, which is used to identify the completion of the execution of a parallel call. In addition, the goals of parallel call, which can be executed by new AND_tasks, are stored in the *Goal Stack*, until being executed by the task itself or by a new task. Finally, the switch *SEQ_A/PAR_A* has been introduced to distinguish between sequential execution and AND_parallel execution.

A scheme of the architecture including these new data structures is depicted in Figure 5.1.

5.2. Instructions to Manage Parallelism

The set of instructions of PDP consists of the WAM instructions, some of them slightly modified, along with instructions to manage each kind of parallelism.

The following instructions are introduced to deal with OR_parallelism:

- *try_par*
- *retry_par*

These instructions replace the *TryMeElse* and *RetryMeElse* instructions, respectively, for the first clause of a set annotated with OR_parallelism. The new instructions have the same semantics as the sequential ones, but they also annotate which a new pending job is prepared for the creation of an OR_task and send a warning to the scheduler. The address of these instructions is also recorded (try_par) or updated (retry_par) in the success path.

The instructions TryMeElse, RetryMeElse and TrustMeElseFail have been modified to record and update the success path.

The analysis and control of the execution of a parallel call is carried out by similar instructions to those designed by Hermenegildo [19] but using an single code for the sequential and parallel execution of the parallel call. This code is interpreted as sequential or parallel depending on the state of the SEQ_A/PAR_A switch. The new instruction *par_exec* changes this switch to the state PAR_A and the execution is performed in parallel. The instructions *check_ground* and *check_independent* perform tests on the groundness and independence of the variables, *alloc_p* allocates space in the Stack for a *Parallel Call Environment*, *pop_goal* executes a local goal and *wait* is used for the synchronization of answers.

For instance, the following code, in which AND and OR_parallelism appear, will be translated as Figure 5.2 shows.

$p(X,Y) : -independent(X,Y) \Rightarrow (q(X)\&q(Y)).$
$*q(X)$:- $r(X)$.
$q(X)$:- $s(X)$.
$q(X)$:- $t(X)$.

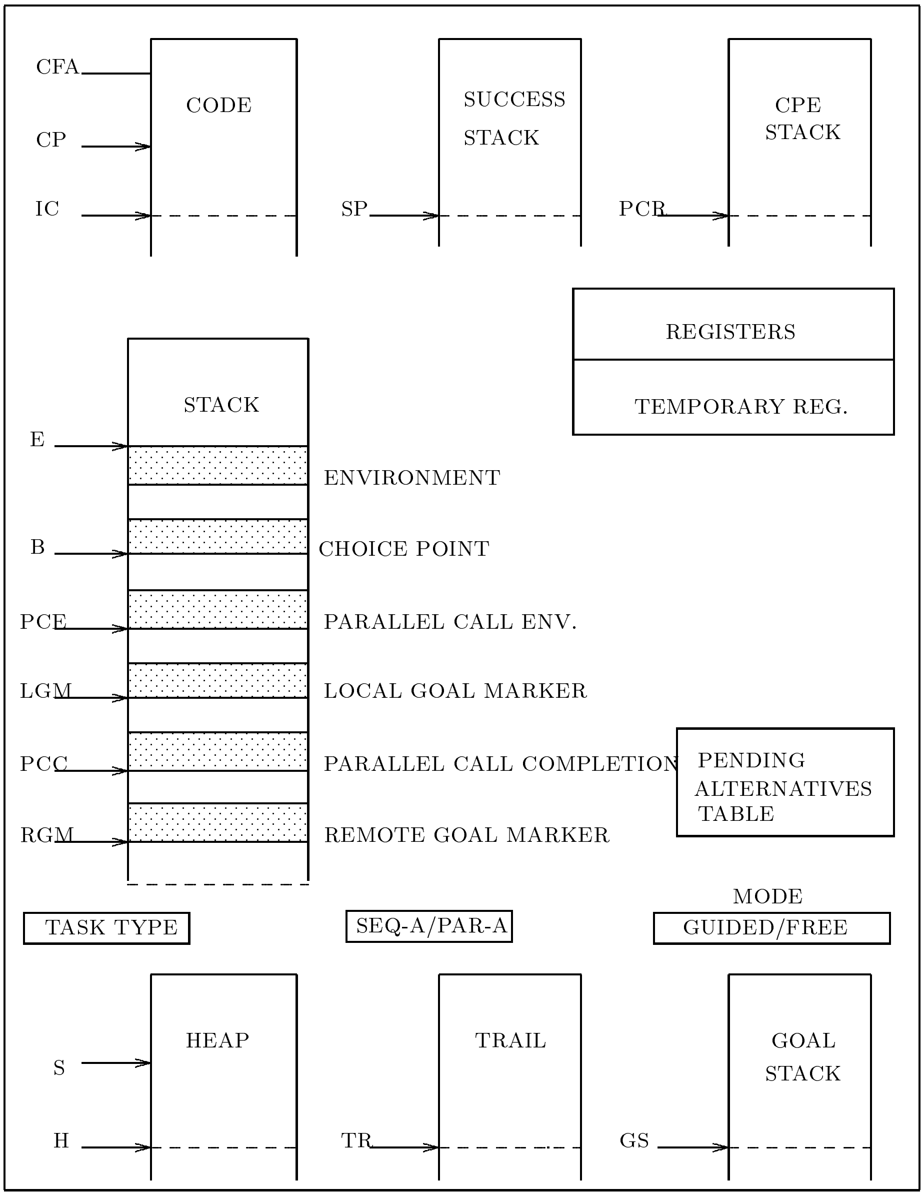

FIGURE 5.1. Data Structures of a Worker

```
p_1 alloc                         q_1 try_par 0 q_2
      get_var r1, X                     call r, 1
      get_var r2, Y                     proceed
      par_exec                    q_2 retry_par
      check_independent X, Y            call s, 1
      alloc_p 2, 2                      proceed
                                  q_3 trustMeElseFail
      put_val X                         call t, 1
      call q_1,0                        proceed

      put_val Y
      call q_1,1

      pop_goal
      wait
      proceed
```

FIGURE 5.2. Compiled Code Example

6. SCHEDULING POLICY

The scheduling of pending work among the workers of the system is an important issue for the overall performance. In fully distributed memory systems, the information needed to carry out the scheduling is usually centralized, in order to reduce the exchange of messages. PDP has been designed with a hierarchical scheduling policy, in which the schedulers are responsible for the distribution of pending work among idle workers. However, in the current implementation, one single scheduler is enough to support the number of available processors. A worker may be in one out of three states: *idle* (without work), *busy* (working) or *offering* (with pending work). The scheduler knows the state of the workers in its cluster. The scheduling policy determines which offering worker has to be requested by which idle worker. To optimize the communications, the workers do not report every change in their workload. The scheduler has exact information about idle workers and offering workers, and approximate information about the workload of the workers.

In order to choose the offering worker which is going to share work with an idle worker, several strategies previously used by different systems [29, 18] 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 6.1 shows the speedup on a system with 15 workers for each strategy. Since the measured times differ from run to run, all given speedups are computed

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 6.1. Speed-up for different scheduling policies

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, i.e. to choose 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, thus optimizing OR_parallelism exploitation. The worst strategy is B, since it is the most expensive one (it requires more information exchange than the other two), while strategy C, i.e. to choose the oldest request, is almost as good as A, because it does not need any analysis by the scheduler.

7. GRANULARITY CONTROL

The parallel execution of a task is worthwhile if its execution time is greater than the time spent to be scheduled for parallel execution. Therefore, a task is scheduled for sequential or parallel execution depending on its granularity [13, 14, 30], i.e. a time estimation of its execution. In PDP, the schedulers distribute the pending work among idle workers, and it corresponds to the workers to decide whether a task has enough granularity to be executed in parallel. PDP applies a granularity control for OR_parallelism exploitation based on heuristic observations concerning memory occupation. In the case of AND_parallelism exploitation, PDP applies a granularity control based on the estimation provided by the system CASLOG [14].

PDP performs a granularity control for the exploitation of OR_parallelism by checking the similarity of the amount of data in the stack when each solution is reached. Thus, it is possible to estimate how close a worker is to finding the next solution and therefore whether it is worthwhile to share the work that leads to that solution. 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 than a threshold value. This threshold depends on the system and it is obtained experimentally. This test does not introduce run time overhead since only a comparison is needed. Figure 7.1 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 threshold value T. This means, as the search tree shows, that $T2$ has fine grain and therefore the task is

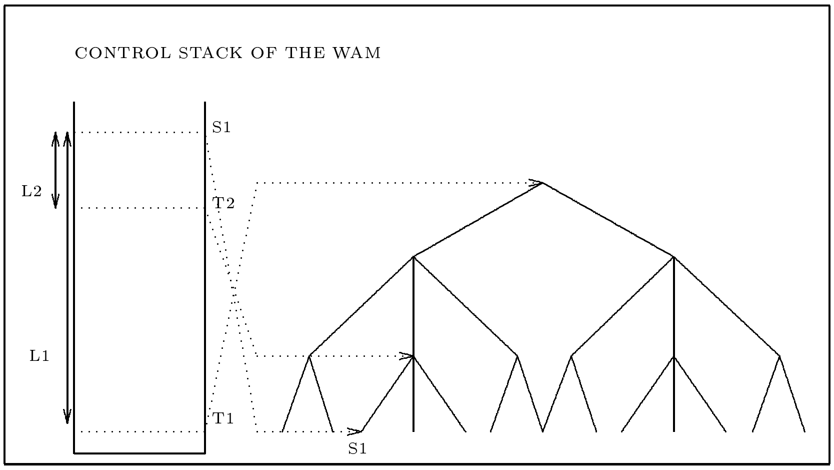

FIGURE 7.1. Granularity estimation of OR_parallelism exploitation

not sent to another worker. On the other hand, the task $T1$, corresponding to a choice point in the bottom of the stack, has high granularity and thus it is sent to an idle worker.

	3 workers		8 workers		15 workers	
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 7.1. Speed-up achieved by controlling OR_parallelism granularity

Measurements have been taken to determine the threshold value T. On a system with 8 to 15 workers, this value (which 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 7.1. 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.

In the case of AND_parallelism, PDP applies a control mechanism based on the estimation provided by the CASLOG system [14] – which is also used by the &-Prolog compiler [24] in order to control the granularity. CASLOG provides in a (semi)automatic manner an upper bound to the cost of a large class of logic programs. This bound – which is tight enough to be used as an estimation of the granularity – is given in the form of an algebraic expression, depending on the size of the arguments. This expression is evaluated at execution time by calculating the size of the arguments. This calculation may be very complex, and therefore, the application of this control can introduce an significant overhead. In PDP, however, this control mechanism is applied at a moment in which the data size is known, that is, when the message of creation of a new AND_task is being prepared (including the goal and the binding of its variables). In this way, the control does not introduce additional overhead. Table 7.2 presents the speedup achieved by introducing this granularity control. The benchmarks are the *merge* and *qsort* programs for sorting and the program *matrix* for multiplying a matrix and a vector. With 4 workers, the control does not have any effect, since the size of the system limits the exploitation of parallelism. The control produces its greatest effect with 8 workers and the programs *merge* and *qsort*. *Matrix* does not exhibit enough parallelism to require this control for this number of workers, and the same occurs for *merge* and *qsort* with 16 workers.

	3 workers		8 workers		15 workers	
program	no c.	with c.	no c.	with c.	no c.	with c.
merge(500)	1.38	1.4	1.78	1.8	1.93	1.94
qsort(700)	1.42	1.42	1.91	2.25	2.25	2.25
matrix(75)	1.38	1.38	1.73	1.73	1.78	1.78

TABLE 7.2. Speed-up achieved by controlling AND_parallelism granularity

8. PERFORMANCE RESULTS

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

Some sequential programs have been run on PDP in order to evaluate the overhead due to the parallel mechanism. The experiments demonstrate that this overhead is at most 10% (see Tables 8.2, 8.5 and 8.9), and it is mainly due to the checking of arriving messages and the recording of the success path (results show that the overhead due to the latter is less than 5%, as we have checked on an implementation in which this mechanism has been isolated). The absolute execution times have been included in Tables 8.2, 8.5 and 8.9. Although these execution times are worse than those of other parallel systems, such as MUSE [2], this is not due to the parallel mechanism. Since MUSE and PDP systems follow a multisequential approach, the time improvements obtained by the optimizations in each sequential thread yield an overall improvement of the parallel execution time. For this reason, the MUSE system, which has been implemented on a sequential Prolog with a high level of optimizations (SICStus), has better execution times that those of PDP, which has been implemented on a sequential system without full optimizations. Table 8.1 compares the execution times of the SICStus system (version 2.1) and the sequential system of PDP, the former running on a SUN *SuperSPARC* and the latter running on both a SUN *SuperSPARC* and a transputer T800. The introduction of further optimizations in the sequential system of PDP will reduce the parallel execution time. However, the speedup (sequential time/parallel time) will in general decrease because there are some components of the program which do not improve with the sequential optimizations, such as link communications. Nevertheless, the speedup is still a meaningful parameter because, given a certain sequential system, it allows to choose between different strategies to design particular aspects of the parallel system.

program	SICStus (ver. 2.1)	sequential (SUN)	sequential (transputer)
merge	451	624	783
qsort	630	915	1185
matrix	779	1001	1271
queen(8)	21310	32133	37245
queen(9)	112830	141296	187043
queen(10)	565810	771361	966818

TABLE 8.1. Evaluation of the sequential system on which PDP has been built.

For each kind of parallelism, we have investigated the speedup achieved and the time distribution of a worker among its activities, namely the following:

- **Execution**
  Time spent in executing its sequential tasks.
- **Inactivity**
  Time in which the worker is idle, waiting for a new job.
- **Recomputation**
  Time spent in recomputing the success path of a new job. This only occurs during the exploitation of OR_parallelism.
- **Waiting**
  Inactivity time due to waiting for the answers of other workers. This time only appears during the exploitation of AND_parallelism.
- **Communications**
  Time spent in communication with the scheduler and other workers. This time includes the time spent in constructing the messages.

We have also studied particular aspects of each kind of parallelism, such as the comparison between the recomputation model and the copying model for OR_parallelism, and the time that is saved due to the avoidance of dereferences if AND_parallelism is exploited.

8.1. OR_parallelism Evaluation

Figure 8.1 shows the speedups achieved by exploiting OR_parallelism in some benchmarks. All benchmarks have been executed in a "program, fail" way, in order to reach every solution. The speedups increase almost linearly with the number of processors, the slope being larger for programs with coarse grain parallelism (such as *queen10*). For the program *chat*, which has a very small grain, the performance increases only slightly and even reaches a saturation. Absolute execution times appear in Table 8.2. The first column shows the times achieved by the sequential system on which PDP has been built. The remaining columns report the times achieved by PDP on different numbers of workers. From the two first columns (the sequential execution and the PDP execution on only one worker) it can be seen that the overhead introduced by the parallel mechanism is about 10%.

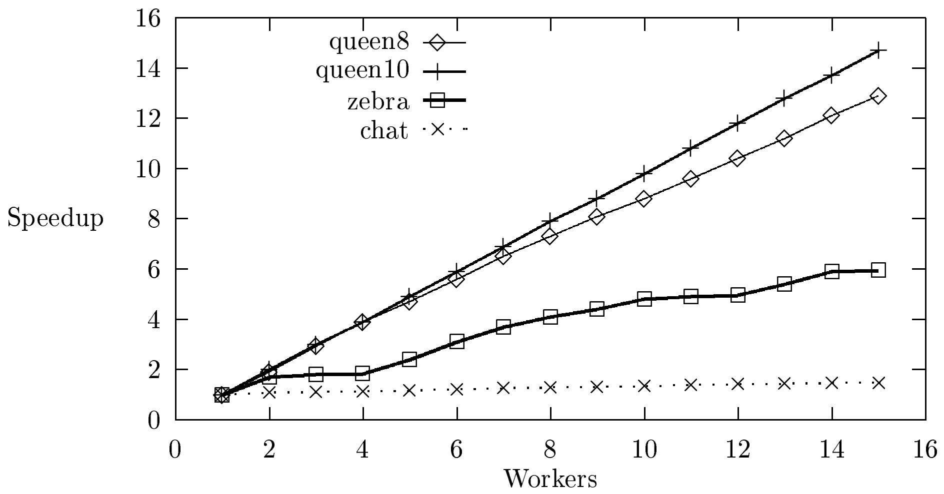

FIGURE 8.1. Speedup achieved by exploiting OR_parallelism

program	sequential	1 worker	3 workers	8 workers	15 workers
chat	206	240	218	218	218
zebra	1096	1160	644	400	293
queen(8)	37245	39040	13943	5348	3026
queen(9)	187043	198889	82870	25829	14006
queen(10)	966818	1028119	331651	130142	69940

TABLE 8.2. Absolute execution times in ms exploiting OR_parallelism. Programs are sorted by increasing grain. To illustrate the overhead introduced by the parallel implementation we have included the executions times of the sequential system on which PDP has been built.

Table 8.3 shows that the system can also reach an important speed-up when only the first solution is required. This can be easily understood if one thinks that the parallel system may find solutions shorter than the first solution a sequential system would find. Nevertheless the best performance of the system is achieved when all solutions are requested.

program	speedup
queen(8)	5.2
queen(10)	9.3

TABLE 8.3. Speedup for single-solution execution with 15 workers.

Time Distribution

We have measured the time distribution of the workers among the activities carried out during OR_parallelism exploitation, in order to draw some conclusions regarding the results obtained. The time devoted to each activity has been measured as the average time of the workers which have taken part in the execution. Table 8.4 presents the percentage devoted to each activity relative to the total time. As the number of workers increases, the percentage of *execution time* decreases and the *inactivity time* increases, since the amount of parallelism of a program is limited. We can observe that the percentage of the *recomputation time* is very small, less than 5% in all cases. This percentage slightly increases with the number of workers, because there is a larger number of tasks and the recomputation is performed more times. The high percentage of the *inactivity time* in the program *chat* indicates the lack of parallelism of this program, and justifies the small speedup.

Program	Activity	3 workers	8 workers	12 workers	15 workers
queen10	Execution	99.85	99.5	99.01	98.17
	Inactivity	0.09	0.27	0.76	1.48
	Recomputation	0.009	0.02	0.03	0.05
	Communications	0.05	0.12	0.18	0.30
queen8	Execution	91.43	91.1	81.81	79.54
	Inactivity	7.29	7.12	15.8	16.37
	Recomputation	0.04	0.14	0.24	0.32
	Communications	1.24	1.64	2.15	3.77
chat	Execution	7.61	6.45	4.32	3.12
	Inactivity	91.9	93.1	95.2	96.3
	Recomputation	0.01	0.01	0.01	0.02
	Communications	0.41	0.43	0.46	0.52

TABLE 8.4. Time distribution (percentage) of a worker in the OR_parallelism exploitation

Comparison with a Copying Approach

We have compared the speedup obtained exploiting OR_parallelism using both stack copying and recomputation (the stack copying implementation follows the MUSE [2] approach, but it has been built on the same sequential system as PDP). The results are shown in Figure 8.2. These results show significant speedup in either approach for all tested programs, though there is very little difference between the two methods. Nevertheless, the speedup is slightly larger for the copying approach when there is a small number of workers and when programs have small granularity, and the speedup is larger for the recomputation approach when the system size increases. The reason is the smaller amount of information exchanged 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. The difference of speedup is important for the *chat* program when the number of workers increases. Similar conclusions have also been obtained with the Eclipse [26] distributed version.

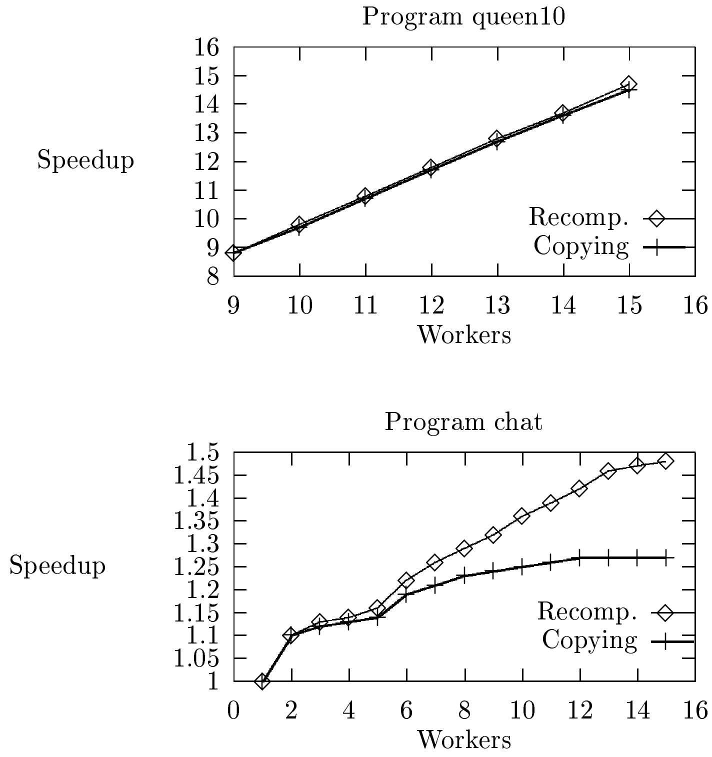

FIGURE 8.2. Comparison of copying and recomputation approaches

8.2. AND_parallelism Evaluation

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. Figure 8.3 shows the speedup achieved for the programs *qsort*, *merge* and *matrix*. For programs with finer grain parallelism, no improvement of speedup is achieved by exploiting AND_parallelism. Results show that the amount of parallelism exploited is small and the execution time does not decrease from 10 workers upwards. Absolute execution times are presented in Table 8.5.

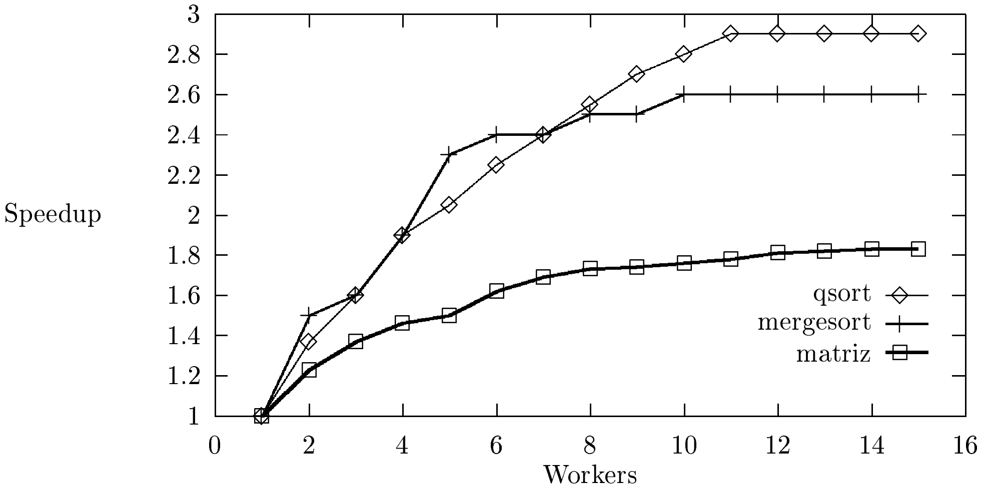

FIGURE 8.3. Speed-up achieved by exploiting AND_parallelism

program	sequential	1 worker	3 workers	8 workers	15 workers
qsort(700)	1185	1289	560	460	379
merge(500)	783	830	360	332	307
matrix(75)	1271	1394	996	820	774

TABLE 8.5. Absolute execution times in ms exploiting AND_parallelism

Time Distribution

Table 8.6 presents the percentage devoted to each activity when AND_parallelism is exploited. We can observe that the percentage of *inactivity time* is high, larger than that obtained in the OR_parallelism exploitation. This is due to the fact that AND_parallelism has smaller granularity than OR_parallelism for the pro-

Program	Activity	3 workers	8 workers	12 workers	15 workers
qsort	Execution	72.04	54.84	42.24	30.09
	Inactivity	14.02	32.23	46.85	61.02
	Waiting	7.93	7.12	6.85	4.39
	Communications	6.01	5.81	4.06	4.5
merge	Execution	43.08	26.16	15.0	12.25
	Inactivity	43.82	62.58	77.78	80.88
	Waiting	6.95	5.12	3.34	1.73
	Communications	6.15	6.14	3.88	5.14
matrix	Execution	39.75	23.75	17.38	12.06
	Inactivity	56.11	72.34	79.50	84.87
	Waiting	0.11	0.11	0.11	0.11
	Communications	4.03	3.8	3.01	2.96

TABLE 8.6. Time distribution (percentage) of a worker in the AND_parallelism exploitation

grams checked. We can also observe that the percentage of *communications time* is larger than that obtained in the study of OR_parallelism, resulting in a smaller improvement of the performance. The percentage of *waiting time* increases along with the percentage of *execution time*, since the parallelism which produces the waiting time is exploited during the execution of the program.

Dereferencing Time

In WAM based Prolog implementations, the binding of a variable to a term consists of a chain of references. Therefore, the unification of a variable may require a dereference to find the term to which the variable is bound. PDP substantially reduces the number of dereferences, since a new AND_task reaches its goal with dereferenced arguments. Thus, these arguments are accessed in a shorter time than in the parent task. Table 8.7 compares the number of dereferences by exploiting AND_parallelism with different number of workers. The percentage of dereferences saved ranges from 7% for the program *merge* to 39% for the program *matrix*.

Program	1 worker	4 workers	8 workers	15 workers
merge (500)	39353	37918	36806	36584
qsort (700)	67462	65285	62267	61120
matrix (75)	28500	21252	17930	17326

TABLE 8.7. Number of dereferences exploiting AND_parallelism

program	OR_par.	AND_par.	Comb. par.
synthetic 1	1.5	2.9	4.5
synthetic 2	2.37	1.17	4.6

TABLE 8.8. OR, AND and Combined parallelism speedup

8.3. Evaluation of Combined Parallelism

PDP is expected to perform better for programs complex enough to present high granularity. Realistic programs usually fulfill this condition, but they exceed the limitation of processors and memory of the current prototype. Therefore, the system has been evaluated with two high granularity synthetic benchmarks for which the performance of PDP is expected to be optimal. 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,Us,Vs]) :- times(V), (p(Xs,Ys) & p(Zs,Us)), r(V,Vs)).
p(Xs,Ys) :- times(X), times(Y), (r(X,Xs) & r(Y, Ys)).
p(Xs,Ys) :- times(X), times(Y), (s(X,Xs) & s(Y, Ys)).
r(X,Xs) :- r1(X,Xs).                          s(X,Xs) :- s1(X,Xs).
r(X,Xs) :- r2(X,Xs).                          s(X,Xs) :- s2(X,Xs).
r1(0,a).                                      s1(0,c).
r1(X,Xs) :- X > 0, X1 is X-1, r1(X1,Xs).      s1(X,Xs) :- X > 0, X1 is X-1, s1(X1,Xs).
r2(0,b).                                      s2(0,d).
r2(X,Xs) :- X > 0, X1 is X-1, r2(X1,Xs).      s2(X,Xs) :- X > 0, X1 is X-1, s2(X1,Xs).

times(1000).
```

There is AND_parallelism in the body of the *check* and *p* clauses, while the procedures *p, r* and *s* present OR_parallelism. Table 8.8 presents the speedup achieved by exploiting each kind of parallelism with 15 workers. Results show significant speedup for every case. 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 slowest 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 always improves. However, these results are limited by the number of transputers of our implementation, which are not enough to exploit all available parallelism in the benchmark 2 (program synthetic 2). 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 certain number of message exchanges between the parent worker and the worker exploring each goal in the parallel call (new solution requests and answers) which are unnecessary when OR_under_AND parallelism is exploited. Table 8.9 compares absolute execution times on a system with 15 workers.

program	sequential	OR_par.	AND_par.	Comb. par.
synthetic 1	740	493	255	164
synthetic 2	9108	3843	7784	1989

TABLE 8.9. Absolute execution times in ms exploiting exploiting OR, AND and Combined parallelism

8.4. Comparison with Other Results

There are several alternative implementations which exploit parallelism of Prolog programs. However, a direct comparison with PDP is difficult, since the results have been obtained for different programs and with a different number of workers. Figure 8.4a) shows a comparison for the execution of the program *qsort*. Although the results are obtained for lists with a different number of elements, a comparison is still worthwhile. In a distributed memory system and exploiting only AND_parallelism, Verden [31] obtained similar results to those of PDP. The system by Hermenegildo [19], with shared memory, produces better results, because shared memory is logically more efficient as it avoids communications.

As concerns OR_parallelism, Figure 8.4b) shows a comparison for the execution of the program *queen8*. The speedup obtained for this program is similar to that obtained by the MUSE system [2]. However, this result could be improved for programs of larger granularity, such as *queen10*.

Several models have also been proposed to combine parallelism. Some of them, such as the PEPsys model [34], the Gupta-Jayaraman model [16], and the ACE model [17], use partially or totally shared memory. The PEPsys model uses a hash-window scheme to deal with multiple bindings. This model treats the combination of independent AND_parallelism with OR_parallelism by means of the formation of cross-products of solution sets for the AND_parallel branches. The solutions are lazily gathered in different virtual bags. The Gupta-Jayaraman model is based on the binding-array method for OR_parallelism and the RAP (Restricted And-Parallelism) method for AND_parallelism. The combined AND\OR model avoids redundant computations by representing the cross-product of the solutions from the AND\OR_parallel goals rather than re-computing it, as PDP does. The ACE model combines the stack copying approach of MUSE [2] and the AND_parallel system of &-Prolog [20]. Teams of processors devoted to the exploitation of AND_parallelism share a same address space.

There also exist models which combine AND\OR_parallelism oriented to distributed memory architectures, such as the Reduce-OR model [21]. This model

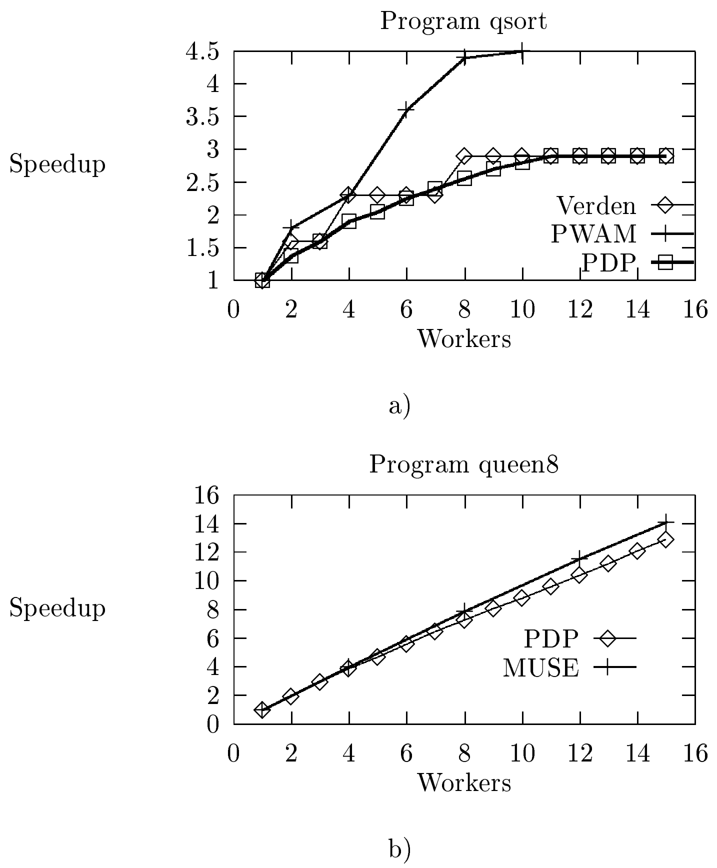

FIGURE 8.4. Comparison with other results

is based on a modified version of the AND-OR tree for the representation of the search tree. The Reduce-OR tree, which consists of Reduce and OR nodes, divides the search tree into independent subproblems. The combination of AND-OR parallelism is solved by using an incremental join operation on solutions tuples from AND-parallel branches. On the contrary, PDP does not perform a join, since each task is indicated the solution to compute in order to solve each goal in a parallel call.

The main goal of the combination of OR\AND_parallelism is the generation of all the combination of solutions to a parallel call. Some of the above mentioned systems do it in a centralized way, storing these solutions into virtual bags, some others generate the solution in an automatic and decentralized way, but this requires an active exchange of messages between processors. The key of the success of PDP lies on the fact that the recomputation mechanism combined with the automatic generation of solutions avoids this communication overhead because it allows the processors work independently of each other. As a result, the combined parallelism performs much better than the OR_parallelism and AND_parallelism together (as illustrated in Table 8.8, where the speedup for the combined parallelism is shown to be larger than the product of the other two separately). Unfortunately, since PDP requires complex programs with high granularity to reach an optimal performance, we can not compare our results with those of the other systems which exploit combined parallelism, because these results are obtained for programs which have not enough granularity.

9. FURTHER EXTENSIONS: INCLUSION OF SIDE-EFFECT PREDICATES

On the inclusion of side-effect predicates in parallel logic programming systems a few works have been reported so far. Some of them follow the philosophy of constraining the exploitation of parallelism only to non conflicting cases. The ideas of some of these methods can be applied to PDP [6]. Thus, the *cut* predicate can be implemented following Ali's [1] approach, which consists in constraining the OR_parallelism exploitation to the cases outside the scope of a *cut*. This conservative approach is best justified on a distributed memory system like PDP in which communications among workers have a higher cost in time than on shared memory systems. *Findall* predicate in PDP can take advantage of the success paths used in the recomputation to perform the ordering of the solutions. The implementation of *input\output* predicates requires synchronization mechanisms to maintain the sequential semantics when AND_parallelism is exploited, such as those proposed by Hermenegildo [28] for the RAP system. This mechanism consists in adding code to the program to synchronize the execution of the side-effects. Since the AND_parallel model of PDP is an extension of the RAP system for distributed memory, the method for the implementation of input\output predicates can be imported to PDP quite directly. Then it is extended for the exploitation of OR as well as OR_under_AND parallelism. The exploitation of parallelism in the presence of *database* predicates can be constrained to some special cases in which the update of the database can be managed efficiently, as it has been pointed out in [25], thus reducing communications.

10. CONCLUSIONS

We have developed a Prolog Distributed Processor (PDP), which exploits Independent_AND\OR parallelism of Prolog on a distributed memory system. The execution model is based on multisequential Prolog engines that work independently under a hierarchical scheduling policy. Independent AND_parallelism is exploited following a *fork-join* approach and OR_parallelism is exploited following a recomputation approach. The amount of data communicated with this approach is smaller than that of the copying approach, which in turn leads to the speed up of the execution. 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.

Important conclusions for the construction of Prolog systems can be obtained. Results show that the overhead introduced when the system has a single worker (sequential execution) is always less than a 10%. OR_parallelism exploitation provides a linear speedup for high granularity programs. The recomputation approach has been shown to give a high performance, which is improved when the system size increases. On the other hand, the exploitation of AND_parallelism also provides a significant speedup for coarse grain programs, though logically less than that obtained in shared memory implementations. 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 exploration of the different solutions when AND_parallelism is exploited requires a certain number of message exchanges between the parent worker and the worker exploring each goal in the parallel call (new solution requests and answers) which thanks to the recomputation mechanism, are unnecessary in PDP when OR_under_AND parallelism is exploited.

Different scheduling policies have been tested. The best results have been obtained when the nearest idle worker is chosen. Granularity controls have been introduced, demonstrating an improved performance when the system size increases.

In our future research we will deal with the optimization of the sequential component of the system, and we will try to extend the system to a larger number of workers as well as to test new applications. It would also be worthwhile to investigate the construction of a mixed system, with some shared memory used for the exploitation of pure AND_parallelism, and devoting the distributed memory to the exploitation of OR_parallelism and combined parallelism. We would also like to apply the PDP approach to other distributed memory platforms such as workstation networks. We are also working in the implementation of side-effect predicates.

Acknowledgments

We are sincerely grateful to the referees, whose reviews and remarks on the first manuscript have significantly improved it and made it more readable.

REFERENCES

1. Ali, K.A.M. *A Method for Implementing Cut in Parallel Execution of Prolog.* Research Report SICS R87001, 1987.
2. Ali, K. A. M and Karlsson, R.. The Muse Approach to Or-Parallel Prolog. *Int. Journal of Parallel Programming 19(2),* pp. 129-162, 1990.
3. Alshawi, H. and Moran, D.B., The Delphi Model and Some Preliminary Experiments. *Fifth Int. Conf. and Symp. on Logic Programming*, MIT Press, pp. 1578-1589, 1988.
4. 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.
5. Araujo, L. and Ruz, J.J. PDP: Prolog Distributed Processor for Independent_AND\OR Parallel Execution of Prolog. *Proc. Int. Conf. on Logic Programming*, MIT Press, pp. 142-156, 1994.
6. Araujo, L. Cut and Side-effects on a Distributed Memory Implementation of Prolog. Technical Report 11-94. Universidad Complutense de Madrid, 1994.
7. Biswas, P., Su S. and Yun D. A Scalable Abstract Machine Model to Support Limited-OR(LOR)/Restricted-AND Parallelism(RAP) in Logic Programs. *Proc. Int. Conf. on Logic Programming*, MIT Press, pp. 1160-1179, 1988.

8. Bueno, F., Garía de la Banda, M. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. *International Symposium on Logic Programming*, MIT Press, pp. 320-336, 1994.
9. Calderwood A. and Szeredi, P. Scheduling Or-parallelism in Aurora - the Manchester scheduler. *Proc. Int. Conf. on Logic Programming*, MIT Press, pp. 419-435, 1989.
10. Chang, J.H., Despain, A. and Degroot, D. AND-parallelism of logic programs based on a static dependency analysis. *Proc. Spring Compton*, IEEE , pp. 218-225, 1985.
11. Conery, J. S. *Parallel Execution of Logic Programs* Kluwer Academic Publisher, 1987.
12. Conery, J. S. Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors. *Int. Journal of Parallel Programming*, 17(2), pp. 125-152, 1988.
13. Debray, S.K., Lin, N.-W. and Hermenegildo, H. Task Granularity Analysis. *SIGPLAN-90 Conf. on Programming Language Design and Implementation*, ACM, pp. 174-188, 1990.
14. Debray, S.K. and Lin, N. *Automatic complexity Analysis of Logic Programs Proc. Int. Conf. on Logic Programming*, MIT Press, pp. 599-613, 1991.
15. Degroot, D., Restricted AND-Parallelism. *Proc. of Int. Conf. Fifth Gen. Comp. Sys.*, ICOT, pp. 471-478, 1994.
16. Gupta, G. and Jayaraman, B., Compiled And-Or Parallelism on Shared Memory Multiprocessors. *North American Conf. on Logic Programming*, MIT press, pp 332-349, 1989.
17. Gupta, G. ,Hermenegildo, M. and Costa, V.S., And-Or parallel Prolog: A recomputation based approach. *New Generation Computing*, 11(3,4), pp. 297-322, 1993.
18. Kuchen, H., Wagener, A. Comparison of Dynamic Load Balancing Strategies. Tech. Report 90-5 . Aachener Informatik-Berichte, 1990.
19. 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.
20. Hermenegildo, M. and Greene, K., The &-Prolog system: Exploting Independent And-Parallelism. *New Generation Computing*, 9(3,4), pp. 233-257, 1991.
21. Kalé, L.V., The Reduce-OR Process Model for parallel Execution of Logic Programs. *J. Logic Programming*, 11, pp 55-84, 1991.
22. Kowalsky, R.A. Predicate logic as a programming language. *Proc. IFIP*, North Holland, pp. 569-574, 1974.
23. Kowalsky, R.A. Algorithm = logic + control. *Communicatons of the ACM*, 22, pp. 424-431, 1979.
24. López García, P. Hermenegildo, M., and Debray, S.K. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computing, Special Issue on Parallel Symbolic Computation*, (in press.), Academic Press, 1996.
25. Mills, J.W., and Buettner, K.A., Assertive Demons. *Proc. Int. Conf. on Logic Programming*, MIT Press, pp. 1403-1414, 1988.

26. Mudambi, S., and Schimpf, J., Parallel CLP on Heterogeneous Networks. *Proc. Int. Conf. on Logic Programming*, MIT Press, pp. 124-141, 1994.

27. Muthukumar, K. and Hermenegildo, M. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. *Proc. North American Conf. on Logic programming*, MIT Press, pp. 166-185, 1989.

28. Muthukumar, K., and Hermenegildo, M., *Complete and Efficient Methods for Supporting Side-effects in Independent\Restricted And-parallelism* Proc. ICLP, pp. 80-97, 1989.

29. Sugie, M. Yoneyama, M. and Tarui, T. Load-Dispatching Strategy on Parallel Inference Machine. *Proc. Int. Conf. on Fifth Generation Computer Systems* , pp. 987-993, 1988.

30. Tick, E. Compile-Time Granularity Analysis for Parallel Logic Programming Languages. *New Generation Computing*, 7, pp. 325-337, 1990.

31. Verden, A. and Glaser, H. Independent AND-Parallel Prolog for Distributed Memory Architectures. Technical Report CSTR 90-17. University of Southampton, 1990.

32. Warren, D.H.D., An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.

33. Warren. D.H.D., Or-Parallel Execution Models of Prolog. *Proc. Int. Joint Conf. on Theory and Practice of Software Development*, Springer-Verlag, pp. 243-259, 1987.

34. Westphal, H, Robert, P, Chassin, J and Syre, J. The PEPsys model: Combining Backtracking, AND- and OR-parallelism. *Proc. Int. Logic Programming Symp.*, pp. 436-448, 1987.