Progress in Transputer and Occam Research
R. Miles and A. Chalmers (Eds.)
IOS Press, 1994

A Transputer-based Prolog Distributed Processor *

Lourdes Araujo
lurdes@dia.ucm.es

Jose J. Ruz
jjruz@dia.ucm.es

Dpto. Informática y Automática
Universidad Complutense de Madrid
Madrid 28040, Spain

Abstract

PDP is a transputer-based multiprocessor for parallel execution of Prolog. The exploitation of parallelism is based on several sequential Prolog engines supported by transputers and working in closed environments. PDP exploits Independent_AND\OR parallelism and deals with OR_under_AND parallelism producing a new computation for each element of the cross product of the solutions. This avoids to store partial solutions and to synchronize processors. The system has been implemented on a torus network of transputers. Different scheduling policies and granularity controls have been tested. Results show that the overhead introduced to sequential execution is quite small. Significant speedup is achieved for coarse grain benchmark programs with each kind of parallelism. For programs presenting AND and OR parallelism PDP achieves better results than the sum of both.

1 Introduction

In order to achieve the performance that is currently being asked of Prolog execution, the development of parallel systems is required. Because of its declarative semantics, Prolog programs present inherent parallelism, being the main kinds *AND_parallelism* and *OR_parallelism*. The former consists of executing in parallel two or more goals in the query or in the body of a clause. The exploitation of AND_parallelism requires taking into account that a logical variable may be bound to different values in the execution of each goal. System exploiting *Independent_AND_parallelism* compute in parallel only the goals being either variable-free (*ground*) or having no intersecting set of variables. Independent_AND_Parallelism has been successfully implemented in the &Prolog system [8] where the parallelism is expressed through the operator "&", which will also be used in this paper, to join the set of independent goals (*parallel call*). Figure 1 shows the trace of sequential and parallel executions of the program appearing in Figure 1a. For the sake of simplicity no variables are considered, but it is

*Supported by the Prontic proyect TIC92-0793-C02-01

supposed that the goals p and r that are parallel processes in Figure 1c are independent. OR_parallelism allows to unify a selected goal with the head of several clauses. If more than one solution is required, in the sequential or AND_parallel executions, they are consecutively reached by backtracking, while in the OR_Parallel execution (Figure 1d) the different solutions are explored in parallel by different processors.

In this paper we present the PDP system, a transputer-based architecture supporting an abstract machine to execute Prolog programs exploiting both, Independent_AND and OR_parallelism. The execution model has been developed as an extension of the Warren Abstract Machine (WAM) [12], the standard sequential implementation technique for Prolog. This allows keeping all optimizations of the sequential Prolog techniques. In order to reduce the communications overhead, PDP has been designed as a multisequential system with hierarchic control. The system is made up of a pool of transputers connected through an interconnection network (Figure 2). Some of them, (*controllers*), support the control functions to perform the distribution of pending work among idle processors. The remaining transputers, (*workers*), work like machines able to execute Prolog programs exploiting parallelism. Each controller is assigned to a *workers* group.

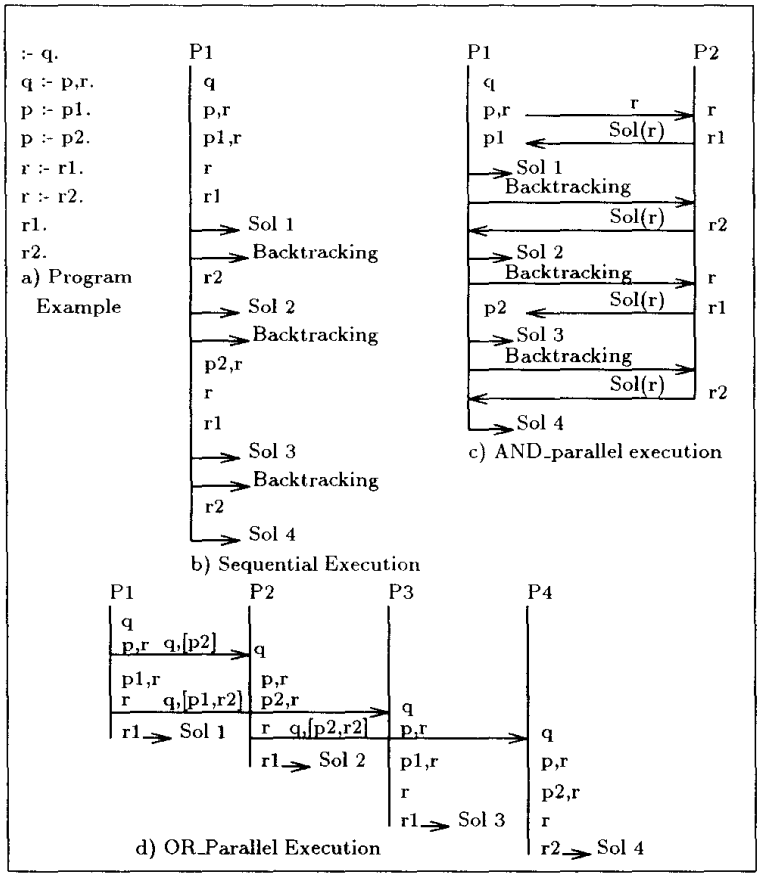

Figure 1: Parallel Execution of a Prolog Program

The PDP model puts together the advantages of exploiting each kind of parallelism. Each PDP worker works in its own environment, following a *closed environment method* [3]: every reference on a worker environment is within this environment. Figure 3 shows the PDP execution for the program in Figure 1a. While in AND_parallel execution

(Figure 1c), the worker P2, that had received a goal of a parallel call, after sending the solution to the parent worker waits for an order to explore the remaining alternatives clauses, in PDP, P2 creates a new computation to deal with those clauses.

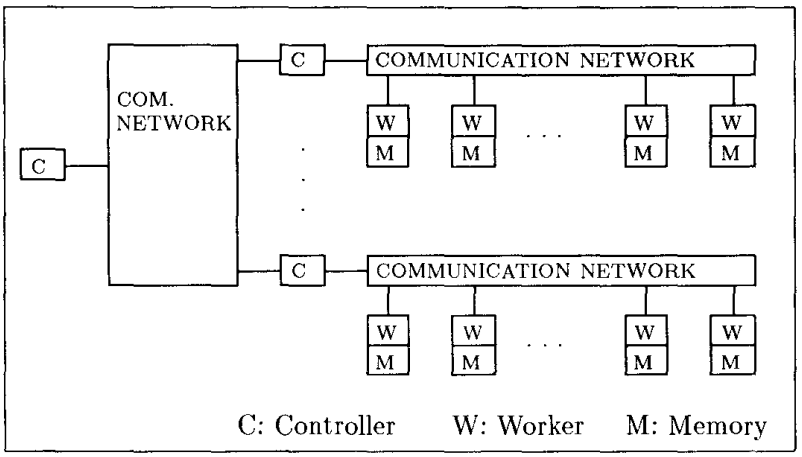

Figure 2: System organization

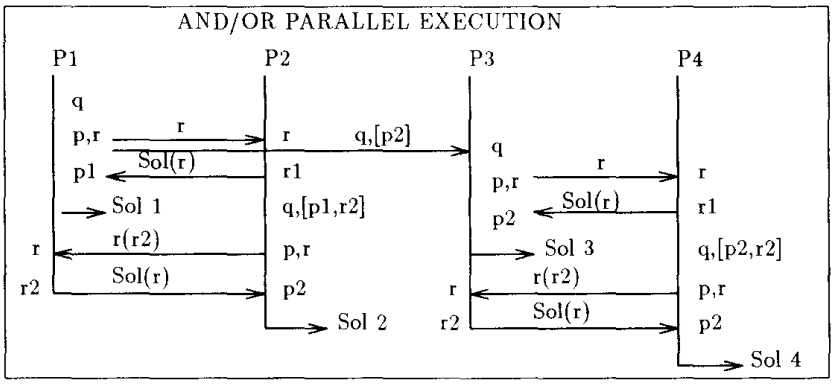

Figure 3: PDP execution

The rest of the paper proceeds as follows: sections 2 and 3 deal with AND and OR parallelism approaches respectively. Section 4 presents the combination approach of both kinds of parallelism. Section 5 is dedicated to the execution model. Section 6 describes the abstract machine of PDP. Section 7 deals with implementation issues. Section 8 describes the scheduling policy. Results are presented in section 9 and finally conclusions drawn in section 10.

2 Exploiting AND_Parallelism in PDP

AND_parallelism speeds up the execution of a parallel call by processing its goals simultaneously. In order to split a parallel call, the PDP worker that finds the parallel call makes a close environment with one of the independent goals along with the binding of its variables (computed answer substitution), sending it to an idle worker. The

parent worker waits for the answer of each goal executed outside. Figure 4 shows an example of an AND_Parallel execution in PDP. Processor 2 receives the goal $p(Y, Y')$, along with the substitution $Y \leftarrow f(b, b)$, and performs the computation independently, sending the computed answer substitution for variable (Y') of the received goal, that is, $Y' \leftarrow f(b', b')$. When the parent processor receives the answer, applies the substitution and continues the execution providing the other goal in the parallel call has been computed.

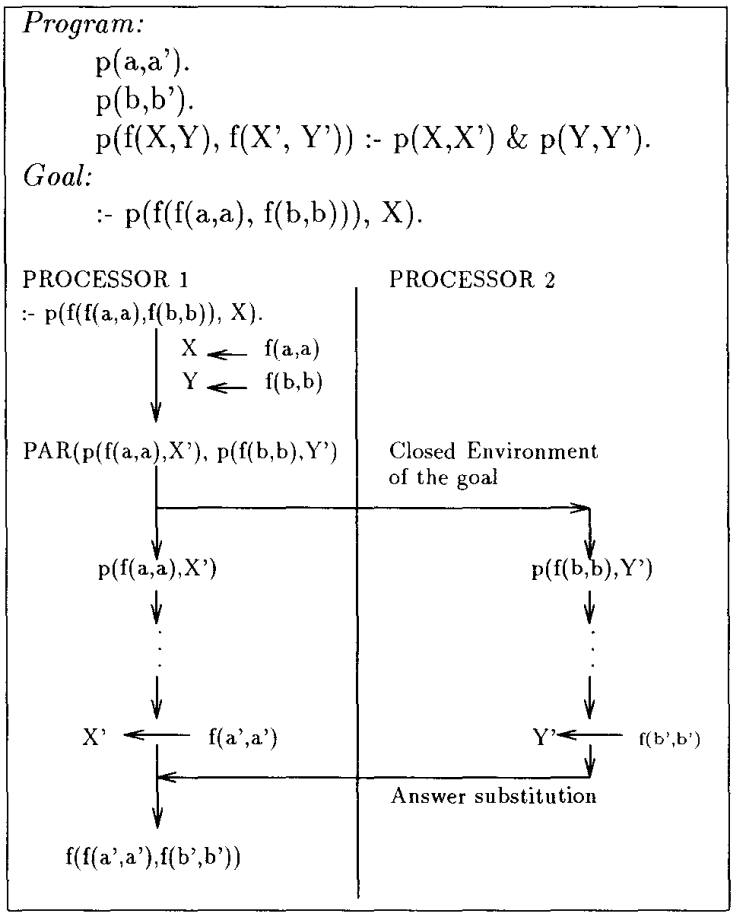

Figure 4: Exploiting AND parallelism

In order to control the execution of a parallel call, we have modified the shared memory model developed by Hermenegildo [7] for a distributed system.

3 Exploiting OR_Parallelism in PDP

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* got from the parent worker, (for example, in Figure 5 the success path is C2, C5, C10). The amount of data communicated

with this approach is smaller than that of the copying approach, what speeds up the execution on a distributed system.

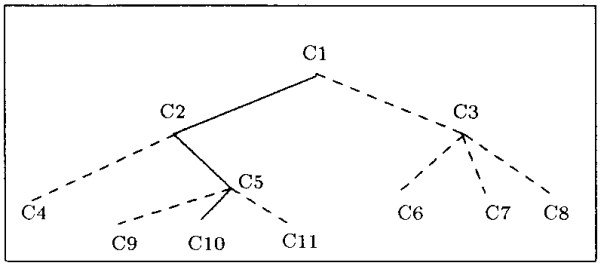

Figure 5: Exploiting OR parallelism: The Recomputation Approach

4 Combining Parallelism in PDP

In a program presenting both kinds of parallelism, depending on the nesting relation between them, two cases can be distinguished. The simpler one represents executions where AND_parallelism appears *under* OR_parallelism. In this case, execution can continue as in the case of pure parallelism since OR_parallel executions give the same machine environment than a sequential execution. 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, taking into account that they have been possibly executed on different processors. In the AND_parallelism approach the task responsible for 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 keeping the AND_parallelism approach requires the storing of the different solutions until being needed to build 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. In this way, the exploitation approach of AND_parallelism becomes that of OR_parallelism, which create independent computations and thus reduces the number of communications. This is the reason why the speedup achieved by exploiting OR and AND_parallelism is greater than the sum of both. Figure 3 shows how the worker P2, given a goal r of a parallel call, performs a recomputation to deal with the clause $r2$ after sending the first solution corresponding to $r1$ to the parent worker. This recomputation corresponds to the combination of alternative clauses $[p1, p2]$. P2 exploits again the AND_parallelism sending r to P1, who has finished its computation. It must be noted that the exploitation of OR_parallelism by P2 could have sent the work to other idle worker instead of keeping it for itself. To avoid repeating solutions, we have introduced a *combination rule* to set the combinations corresponding to each new task.

5 Execution Model

PDP exploits pure AND_parallelism, pure OR_parallelism and the combination of them, producing *OR_tasks*, which explore the search tree branches from the root, and *AND_tasks*

which execute parallel goals. 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 find OR_parallelism, new OR_tasks are created to deal with the parallel clauses. In this way, **the exploitation approach of AND_parallelism becomes that of OR_parallelism**, what reduces the exchange of information. The new OR_tasks need to know the success path leading to the parallel call. This path is given by the parent AND_task which has got it from its parent task.
- If OR_under_AND parallelism is exploited, **it is avoided repeating solutions by introducing a rule to distribute the combinations of solutions among the new OR_tasks.** The *ancestor goal* of a task is the parallel goal whose OR_parallelism has caused the task. The rule fixes the solution to explore for the goals on the left of the ancestor goal and combines them with every solution of the remaining goals. The *combination rule* giving the branch to be explored to solve each parallel goal is as follows:
  - If the goal is on the left of the ancestor goal, the branch to explore is the same explored by the last ancestor OR_task.
  - If the goal is the ancestor one, the branch to explore is the next one to the explored by the AND_task
  - If the goal is on the right of the ancestor goal, the branch to explore 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 to the beginning of the success path corresponding to each goal in a parallel call.

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

- **Primary OR_task:**
  It 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:**
  It 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:**
  It arises from the AND_parallelism exploitation in an AND_task, a primary_OR_task or a secondary_OR_task if it does not correspond to a goal on the left of the ancestor goal. The OR_parallelism that appears during the computation is exploited.

- **Secondary AND_task:**
  It arises from the AND_parallelism exploitation in a secondary_OR_task corresponding to a goal on the left of the ancestor goal.

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 6 shows a PDP execution example corresponding to the following program.

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

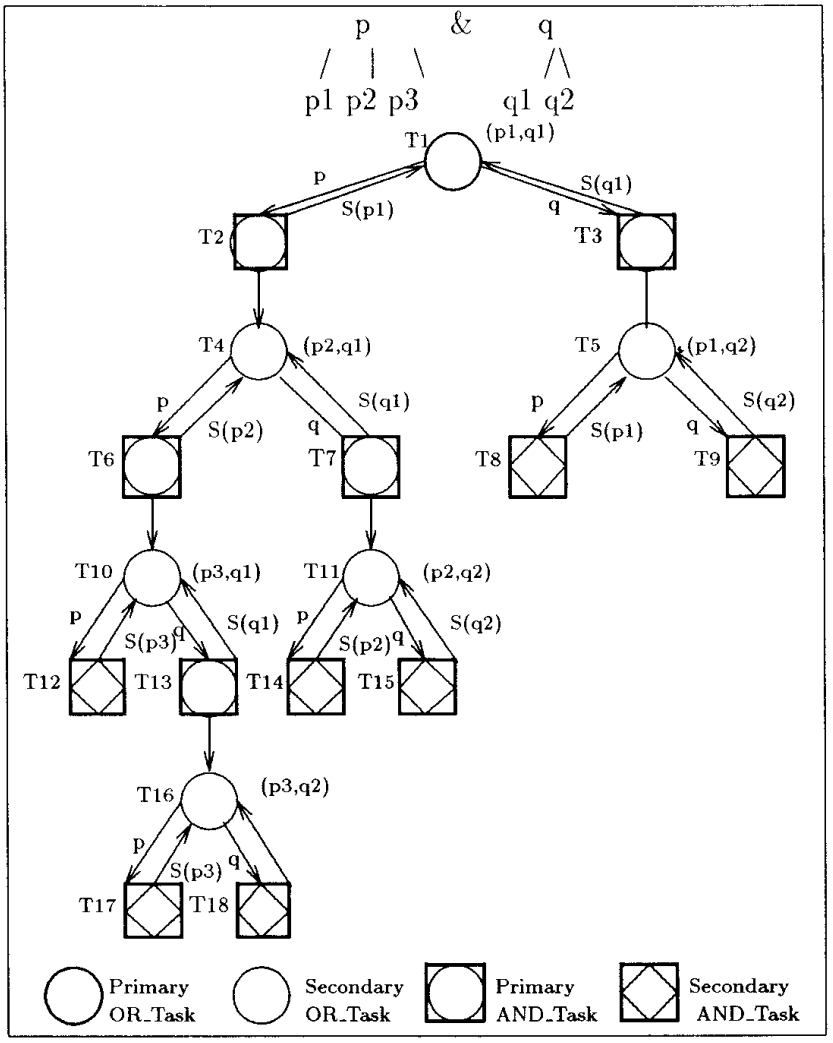

Figure 6: PDP execution example

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 get the solutions, corresponding to the cross product of the 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_tasks T4 for exploring a new solution. T1 builds the CPE $(p1, q1)$, that is passed to N2 and N3. T2 builds the CPE $(p1*, q1)$ because the ancestor goal is the first one (*). This CPE is passed to T4 that builds a new combination taking the following clause, $p2$, for the goal marked as ancestor one, and the first one for the goals on the right, $q1$. T5 receives $(p1, q1*)$, indicating that the alternative clause to exploit for p is the first one and for q it is the second one.

6 Abstract Machine of PDP

A PDP worker, that may be assigned any pending *task*, consists of the WAM along with the extensions to exploit parallelism, that are concerning to the following points:

- *Fork and join control of a parallel call*
  The parallel execution of a parallel call requires to synchronize the reception of the answers. The identifier of the worker where each goal is executed is also recorded in order to deal with backtracking properly.

- *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 tried alternative clauses for each goal of a parallel call.

- *Recomputation*
  The workers that have received the task of exploiting OR_parallelism, are able to follow the path that they receive. To explore the parallel call appearing during the recomputation the received CPE list is consulted. The recomputation is optimized by taking advantage of the common parts of the environments of the workers sharing the work, avoiding the recomputation of this part.

- *Behavior depending on the kind of assigned task*
  If it is a secondary AND_node parallelism is not exploited. If it is a secondary OR_node, after the recomputation a new combination of solutions is made up.

These extensions have been implemented with a low overhead for the sequential execution as well as for each kind of parallelism when it appears alone.

7 Implementation on a Transputer Network

Our current implementation has been made in Parallel ANSI C on a Supernode (Parsys) with 16 T800 transputers connected in a torus network (Figure 7). There is a transputer devoted to the input/output functions (IO) directly connected to the host. The controller is placed on a transputer adjoining to the input/output one, in order to receive and send the messages of the user. The remaining transputers implement PDP workers, executing Prolog programs.

In each processor the computation and communication functions have been split. There are three processes controlling the input, output and computation respectively as Figure 8 shows.

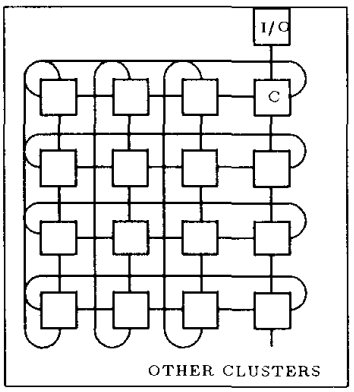

Figure 7: Transputer Network

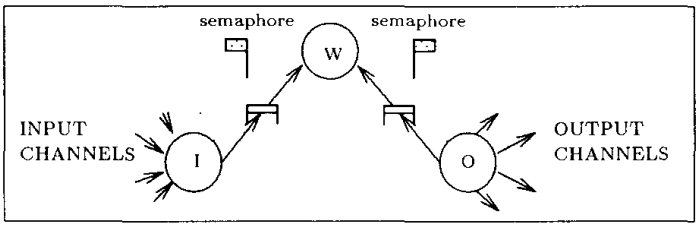

Figure 8: Processes Scheme

The input process waits (*ProcAltList* instruction) the arriving of messages by any of the input channel to the processor. The messages are stored in a global memory area (int_message) whose access is controlled by a semaphore (int_sema).

```
while (1)
{
  Index = ProcAltList(Input);
  SemWait(int_sema);
  message_reception(Input[Index], int_message);
  SemSignal(int_sema);
}
```

The process dedicated to interpret the WAM code of the Prolog programs executes instructions until being warned of an arriving message, that are taken from the *int_messages* area to be analyzed. When it is needed to sent a message to another processor a message is sent by an internal channel to awake the output process.

```
while(1)
{
  if thereis_message
    treat_message();

  execute_instruction(opcode);
}
```

The output process waits to be awaked (*ProcAlt* instruction) by a message sent by an internal channel when a message has to been sent to another processor. The output

channel is determined by the *routing* procedure that decides depending on the destiny address and the occupation of the channels.

```
while(1)
{
  ProcAlt(internal_channel);
  routing(processor_destiny, channel);
  SemWait(out_sema);
  sent_message(out_message,channel);
  SemSignal(out_sema);
}
```

Because of we are working on a regular network, routing is performed in each node using a fixed algorithm based on the local and destination addresses. The routing procedure forwards the message in the direction that reduces the difference between the x- or y- coordinates of the current (*nwam*) and destination (*destiny*) nodes.

```
router(nwam,destiny, output)
char nwam; char destiny; char *output;
{
  char rowc;   char colc;  /* current row and column */
  char rowd;   char cold;  /* destiny row and column */
  char difcol; char diffil;/* difference between columns and arrows */

  /* NW: number of transputers per row or column */
  rowc = nwam / NW;  colc = nwam % NW;
  rowd = destiny / NW; cold = destiny % NW;

  /* checking if they are in the same row */
  if (rowc == rowd) {
    difcol = cold - colc;
    if (difcol > 0) {
      if (difcol <= NW/2)
        *output = EAST_CHANNEL;
      else
        *output = WEST_CHANNEL;}
    else {
      if ((NW + difcol) <= NW/2)
        *output = EAST_CHANNEL;
      else
        *output = WEST_CHANNEL;}}
  else {
    /* the same for NORTH and SOUTH */
}
```

8 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

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

and offering workers telling to the idle workers what offering worker has to be requested for work. To optimize the communications, the workers do not report every change in its 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 going to share work with an idle worker, several strategies previously used by different systems [9, 11, 6] have been tested.

- strategy A: Choose the nearest 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 16 processors 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 well-known, 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 processors having sharing work previously, what optimize OR_parallelism exploitation. The worst strategy is B, since is the most expensive one, while strategy C, choosing the oldest request is almost as good as A, since it is the cheapest one and doesn't need any analysis in 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 [4, 5, 10] when deciding whether or not to execute a job as a separate task. PDP applies control mechanisms based on heuristic observations about memory occupation and waiting time.

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 close is a worker to reach 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 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 9 shows the stack size when solution $S1$ is reached. This point is very close to the choice

point associated to the pending task $T1$ and therefore the distance $L1$ is smaller than the critical value C. That means, as the search tree shows, that $T1$ has a fine grain and therefore the task is kept in the worker. On the other hand, the task $T2$, that corresponds to a choice point in the bottom of the stack, has a high granularity and is sent to an idle worker.

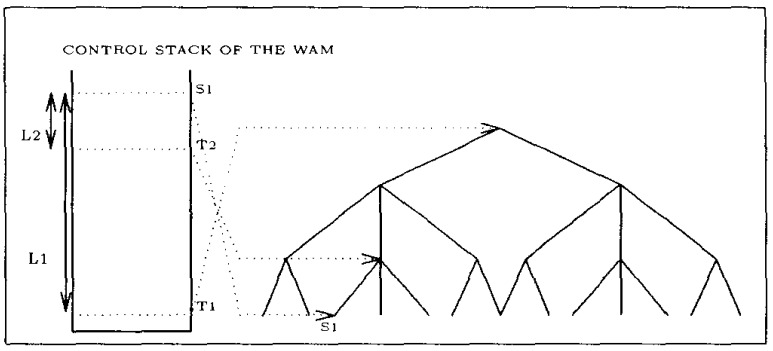

Figure 9: Granularity estimation of OR_parallelism exploitation

Measurements have been taken to investigate the critic value C. On a system with 8 and 16 processors, 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 performing this mechanism is shown in Table 2. For 4 processors the granularity control has not effect. For 8 processors the speedup increases for programs with a lot of parallel work (queen10). The greater effect of the control is achieved on the system with 16 processors.

	4 proc.		8 proc.		16 proc.	
program	without c.	with c.	without c.	with c.	without 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

Another control of the granularity is based on observing that the jobs with coarsest grain are usually close to the root of the search tree. Therefore it is possible to estimate that a job has smaller or equal granularity than the previous one. The performance may be improved if the workers that have had to wait for the answer to parallel goals executed outside after finishing their job, do not exploit AND_parallelism again. This mechanism does not introduce any overhead in the run time. Table 3 shows the result of including this control.

program	4 workers		8 workers		16 workers	
program	without c.	with c.	without c.	with c.	without c.	with c.
qsort	2.3	2.3	2.6	2.8	3.1	3.4
merge	2.2	2.3	2.4	2.5	2.6	2.7

Table 3: Speed-up achieved controlling AND_parallelism granularity

9 Performance Results

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 run on PDP is less than 5%. Results show that the overhead due to write down the success path is less than 5%.

We have compared speedup obtained exploiting OR_parallelism using stacks copying and recomputation approaches. The results obtained are shown in table 4. These results show significant speedup for all tested programs. Though the achieved speedup using stacks copying and recomputation is quite similar, it is slightly better for the copying approach for a smaller number of processors and for programs with smaller granularity and becomes better for the recomputation approach when the system size increases. The reason is the smaller amount of information exchange in the recomputation approach. For queen10 the copying approach is better on a 4 or 8-transputers system, but on a 16-transputers system the recomputation approach becomes better.

program	4 workers		8 workers		16 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 4: Speed_up achieved exploiting OR_parallelism with copying and recomputation approaches

The exploitation of AND_parallelism in PDP requires high granularity programs, since the processors sharing a job exchange more messages that in the case of OR_parallelism. Table 3 shows the speedup for mergesort and qsortsort programs. 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)).
```

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: Speed-up achieved exploiting OR, AND and Combined Parallelism

```
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 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. Results show significant 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 the performance when both kinds of parallelism are exploited has been improved in all cases. The speedup achieved when exploiting both kinds of parallelism is greater than the sum of the speedups achieved by exploiting each kind of them 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.

10 Conclusions

PDP is a transputer-based multiprocessor for parallel execution of Prolog. The execution model is based on multisequential Prolog engines that work independently under a hierarchic control. PDP exploits both, Independent_AND and OR_parallelism. To exploits AND_parallelism parallel goals are sent along with their variable bindings, to the

idle worker indicated by the controller and an answer is waited by the parent worker. The exploitation of OR_parallelism is based on reconstructing the parent processor environment recomputing the success path. 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 to store partial solutions and to synchronize workers. The system has been implemented on a torus network of transputers. 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. Granularity controls have been introduced for both kinds of parallelism, showing a performance improvement when the system size increases. Significant speedup is achieved for coarse grain benchmark programs with each kind of parallelism. For programs presenting AND and OR_parallelism PDP achieves better results than the sum of both. It is because the mechanism to deal with combined parallelism reduces the number of messages exchanges. In the future the system will be extended to a larger number of processors and and new network configurations will be tried.

References

[1] Khayri A. M. Ali and Roland Karlsson. The Muse Approach to Or-Parallel Prolog. International Journal of Parallel Programming Vol. 19 No. 2 April 1990.

[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] Conery, J., Binding environment for parallel logic programs in non-shared memory multiprocessors. In proc. of the 4th. Symp. on Logic Programming, pp. 457-467, San Francisco, 1987.

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

[5] Debray, S.K., Lin, N. *Cost Analysis of Logic Programs* Proc of 8th ICLP. Paris, June 1991.

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

[7] 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).

[8] Hermenegildo, M. and Greene, K.J., &-Prolog and its performance: Exploiting Independent And-Parallelism. In porc. of the 7th Int. Conf. on Logic Programming, 1990, pp. 253-268.

[9] 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.

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

[11] Tubella, J., Gonzalez, A. *Measuring Scheduling Policies in Pure OR-Parallel Programs* PRODE'93 Sep. 1993, Blanes, pp. 57-71.

[12] Warren, D.H.D., An Abstract Prolog Instruction Set. Technical Note 309, SRI International, (1983).

Progress in Transputer and occam Research

edited by

Roger Miles
(University of the West of England, Bristol)

and

Alan Chalmers
(University of Bristol)

WoTUG-17
Proceedings of the 17th World occam* and
Transputer User Group
Technical Meeting
10th–13th April 1994
Bristol, UK

IOS Press
1994
Amsterdam • Oxford • Washington • Tokyo

ISBN 90 5199 163 0
ISSN 0925-4986
Library of Congress Catalog Card Number: 94-075912

Publisher:

IOS Press
Van Diemenstraat 94
1013 CN Amsterdam
Netherlands

Sole distributor in the UK and Ireland:

IOS Press/Lavis Marketing
73 Lime Walk
Headington
Oxford OX3 7AD
England

Distributor in the U.S.A. and Canada:

IOS Press,Inc.
P.O. Box 10558
Burke, VA 22009-0558
USA

Distributor in Japan:

Kaigai Publications, Ltd.
21 Kanda Tsukasa-Cho-2-Chome
Chiyoda-Ku
Tokyo 101
Japan

Printed in the Netherlands