ELSEVIER

Future Generation Computer Systems 13 (1997/98) 421–441

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

Parallel execution of Prolog with granularity control

Lourdes Araujo *, Jose J. Ruz [1]

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

Received April 1995; received in revised form July 1996; accepted August 1997

Abstract

This paper presents a system for parallel execution of Prolog supporting both independent conjunctive and disjunctive parallelism. The system is intended for distributed memory architecture and is composed of a set of workers with a hierarchical structure scheduler. The execution model has been designed in such a way that each worker's environment does not contain references to terms in other environments, thus reducing communication overhead. In order to guarantee the improvement of the performance by the parallelism exploitation, a granularity control has been introduced for each kind of parallelism. For conjunctive parallelism PDP applies a control based on the estimation provided by CASLOG. The features of the system allow to introduce this control without adding overhead. For disjunctive parallelism PDP controls granularity by applying a heuristic-based method, which can be adapted to other parallel Prolog systems. Different scheduling policies have also been tested. The system has been implemented on a transputer network and performance results show that it provides a high speedup for coarse grain parallel programs. © 1998 Elsevier Science B.V.

Keywords: Logic programming; Conjunctive parallelism; Disjunctive parallelism; Granularity; Distributed memory architecture

1. Introduction

Prolog technology has now proven its worth in a wide range of applications such as finance [5], planning [6] and engineering [14,15]. The new software technology will increasingly provide industry with cost effective ways to manage resources, and to generate highly efficient production schedules. Benefits experienced by major users include increased productivity and return on investment, interactive facilities which allow users to generate more efficient production schedules, and user-defined constraints which allow for the accumulation of vital knowledge and experience of expert users.

In order to reach the performance required for real applications, it is possible to exploit the implicit parallelism of Prolog programs by evaluating the multiple alternatives of a goal in parallel (disjunctive parallelism), or by simultaneously executing the goals in the body of a clause (conjunctive parallelism).

* Corresponding author. E-mail: lurdes@dia.ucm.es.

[1] E-mail: jjruz@dia.ucm.

PII S0167-739X(97)00025-3

When the goals in the body of a clause share variables, their execution must be coordinated in order to avoid the same variable being bound to different values. In order to deal with this issue, Degroot [13] 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 which expresses potential conjunctive parallelism. Then the run-time overhead of detecting binding conflict is substantially reduced by having only to check the conditions of the CGEs. Hermenegildo [17] proposed a WAM-based implementation for *conjunctive parallelism* exploitation in shared memory systems, which incorporates Degroot's approach, including backward computation.

In *disjunctive parallelism*, every goal is independent, and a consistency check for shared variables is not needed. However, if the implementation assumes the same memory space for the computation of different solutions (Prolog is non-deterministic) it is necessary to represent different bindings of the same variable. A number of models have been proposed for this. For instance, the SRI model [24] (adopted by Aurora system [8]) keeps the multiple variable bindings in a binding array, and the Argonne model [25] uses a hash array. Systems such as MUSE [1], with a distributed memory space for each parallel execution, do not need a representation for the multiple bindings of a variable. This system transfers an explicit copy of the data of a process to build the environment of the new process.

For programs presenting *disjunctive under conjunctive 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 the *conjunctive disjunctive parallel execution* [7,16,25]. Most of these approaches are implemented on systems with totally or partially shared memory. On contrary, our study has been focused on distributed memory systems, which, in spite of introducing some communication overhead, present other attractive features such as a higher scalability and a more widespread use (workstation networks, for instance). It is worth emphasising that the parallel execution does not always improve the performance, since the mechanism for parallelism exploitation introduces an overhead which can exceed the benefits of performing actions simultaneously. It is only if the grain of parallelism is large enough that its exploitation pays. Therefore, a granularity control must be introduced.

In this paper we present the Prolog distributed processor (PDP), a multisequential system supporting both *independent conjunctive* and *disjunctive parallelism*. PDP exploits the parallelism annotated in the source program in such a way that the sequential semantics is maintained. It introduces as well granularity controls for each kind of parallelism. For conjunctive parallelism PDP uses the CASLOG system (complexity analysis system for logic) [12], which provides an upper bound to the cost of a large class of logic programs. The estimation given by this system has been successfully applied to shared memory parallel systems such as &-Prolog [17]. However, in distributed memory systems, like PDP, this control mechanism is better suited because it does not introduce additional overhead. The data size, needed for CASLOG to evaluate the granularity, is computed when the message of a new *conjunctive task* is being prepared. For disjunctive parallelism PDP controls granularity by using a method based on the assumption, largely confirmed, that disjunctive programs present their solutions approximately at the same level in the search tree. Even though this granularity control has been specifically designed for PDP, it can be adapted to other Prolog parallel systems.

The rest of the paper proceeds as follows: Section 2 introduces the execution model. Section 3 presents granularity controls. Section 4 describes the scheduling policy. Section 5 outlines the implementation of the system. Section 6 illustrates the performance results, and finally conclusions are drawn in Section 7.

2. The PDP system

PDP is a system for the parallel implementation of logic programs conceived for distributed memory architectures. As it is shown in Fig. 1, the system is composed of a pool of processors, organised as a set of clusters (C), each of them consisting of a scheduler (S) and a set of workers (W). Every goal in the system is resolved as a task,

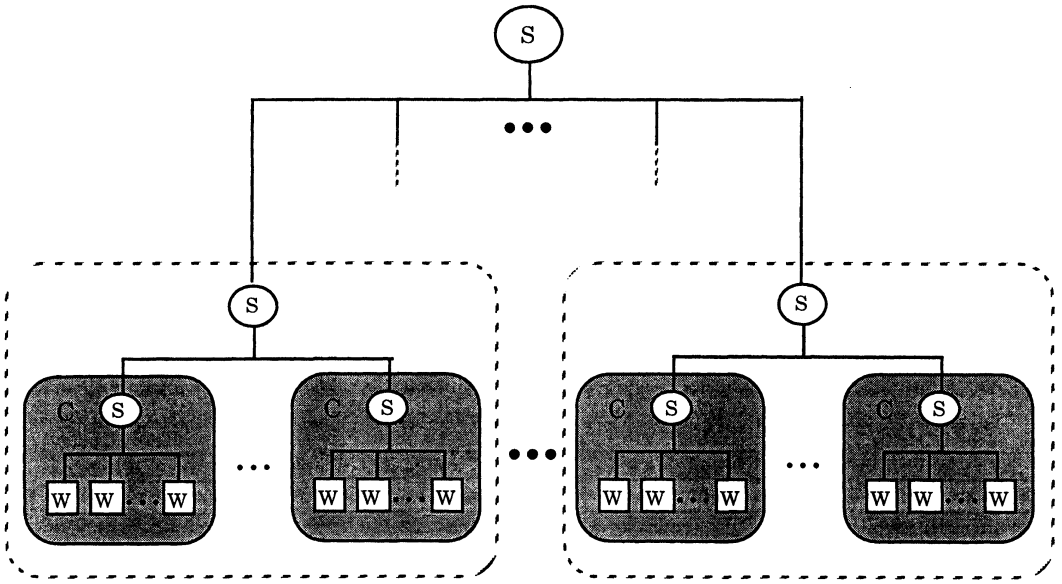


Fig. 1. PDP general structure.

whose execution is supported by a worker – an extension of the sequential Warren abstract machine (WAM) [23]. A copy of the program is loaded in each worker of the system, and the initial goal is assigned to one of them, which becomes the starting worker of the program. Schedulers are responsible for distributing among idle workers the parallel tasks emerged in a worker. When every worker in a cluster is busy, the parallel tasks emerged in the cluster may be sent to other clusters. A higher level scheduler will take charge of the distribution of pending tasks among different clusters. In this way, the scheduling policy has a hierarchical structure whose 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. To reduce the overhead due to communication, the environment of each worker does not contain references to terms in other environments.

2.1. Parallelism identification

The system exploits parallelism annotated in the program either by a precompiler [9,18,20] or by the user himself. *Conjunctive parallelism* is annotated by the & operator placed among the independent goals (*parallel call*). For instance, in the next program to flatten a list, the goals flatten in the first clause can be executed in parallel:

```
flatten(X, [X]): -atomic(X), X\ == [], !.
flatten([], []).
flatten([X|Xs], Ys): -(flatten(X, Ys1)&flatten(Xs, Ys2)), append(Ys1, Ys2, Ys).
append([], L, L).
append([X|L1], L2, [X|L3]): -append(L1, L2, L3).
```

The parallel execution may depend on conditions of groundness or independence of variables appearing in the parallel call. The conditional built-in predicates *ground(Lg)* and *independent(Li)* determine at execution time the groundness and the independence of variables in the lists Lg and Li, respectively. For instance, in the next program,

which performs the symbolic differentiation of an expression, the bodies of the three last clauses are executed in parallel only if the variables U and V are independent, i.e. *independent*([U, V]) is true; otherwise they are executed sequentially.

$d(X, X, 1)$: -var(X).
$d(C, X, 0)$: -atomic(C).
$d(+(U, V), X, +(DU, DV))$: -independent([U, V]) $\rightarrow$ ($d(U, X, DU)$& $d(V, X, DV)$).
$d(-(U, V), X, -(DU, DV))$: -independent([U, V)] $\rightarrow$ ($d(U, X, DU)$& $d(V, X, DV)$).
$d(*(U, V), X, +(*(DU, V), *(DV, U)))$: -independent([$U$, V]) $\rightarrow$ ($d(U, X, DU)$& $d(V, X, DV)$).

Disjunctive parallelism allows the parallel execution of clauses belonging to the same procedure. In PDP *disjunctive parallelism* is exploited by simultaneously executing chunks of clauses of a procedure. A chunk, which consists of one of the several consecutive clauses, is annotated by placing the $*$ operator on the left of the first clause. For instance, in the next program, which verifies if two nodes of a tree are connected, the solutions corresponding to each clause of *get_ branched* are simultaneously explored, and the first chunk of clauses of node are executed in parallel with the second chunk.

*get_branched(X, Y): -connected(X, Y).
get_branched(X, Y): -connected(Y, X).

connected(X, Y): -node(X, Y).
connected(X, Z): -node(X, Y), connected(Y, Z).

*node(a, b).
node(b, f).
...
node(c, h).
node(d, g)
...

In general, a procedure annotated with disjunctive parallelism is expected to produce complex enough computations so as to deserve parallel execution. However, a user who knows the procedural behaviour of the program may decide to execute sequentially a procedure annotated with disjunctive parallelism when it is invoked to solve a particular goal. Thus, the system provides the way to disable the exploitation of parallelism in the execution of a particular goal. Similarly, it is possible to disable the exploitation of conjunctive parallelism.

2.2. Execution model

In order to maintain the optimisations of the sequential Prolog, the execution model has been developed as an extension of the WAM [23]. The system follows a multisequential approach which allows to exploit conjunctive and disjunctive parallelism simultaneously.

2.2.1. Conjunctive parallelism

Conjunctive parallelism is exploited by following a fork-join approach [3]. A worker having to compute the parallel call ($c1$&$c2$& ... &cn) creates a task for each ci. These tasks are assigned to idle workers by the controller. After computing the tasks, the workers return the result to the parent worker. Fig. 3 shows a diagrammatic drawing of the control flow of the program of Fig. 2. Worker $W1$ starts the program execution. After the initial goal

```
c1:  p ( a, b ).
c2:  p ( c, d ).
c3:  p ( f ( X, Y ), f ( Z,  T )) :- p ( X, Z) & p ( Y, T ).

goal:

     ? p ( f ( f ( a, a ), f ( c, c ))),  U ).
```

Fig. 2. Program with conjunctive parallelism.

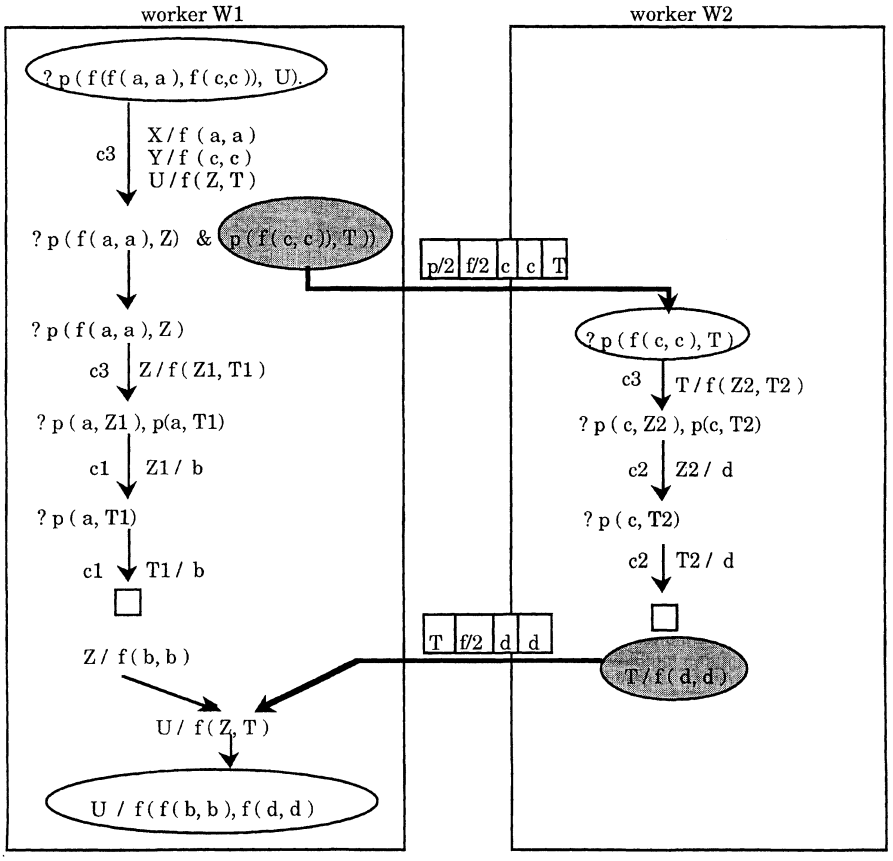


Fig. 3. Conjunctive parallelism exploitation.

$p(f(f(a,a), f(c,c))), U)$ is resolved with the clause $c3$, it appears the parallel call $p(f(a,a), Z)$ & $p(f(c,c), T)$. The independent goal $p(Y, T)$ is scheduled on worker $W2$, so worker $W1$ builds a message consisting of the goal and its bounded arguments to be sent to $W2$. This goal is executed in an autonomous way, and the result is sent back to worker $W1$.

```
c1:  p (X, Y) :- q (X), r (Y).
c2:  q (X) :-  s (X, L).
c3:  s (a,  b).
c4:  *r (Y) :- t (Y, a).
c5:  *r (Y) :- u (b, Y).
c6:  *r (Y) :- v (Y, Y).
c7:  t (X, Y) :- ...
c8:  u (X, Y) :- ...
c9:  v (X, Y) :- ...

goal:

     ? p (a, U).
```

Fig. 4. Program with disjunctive parallelism.

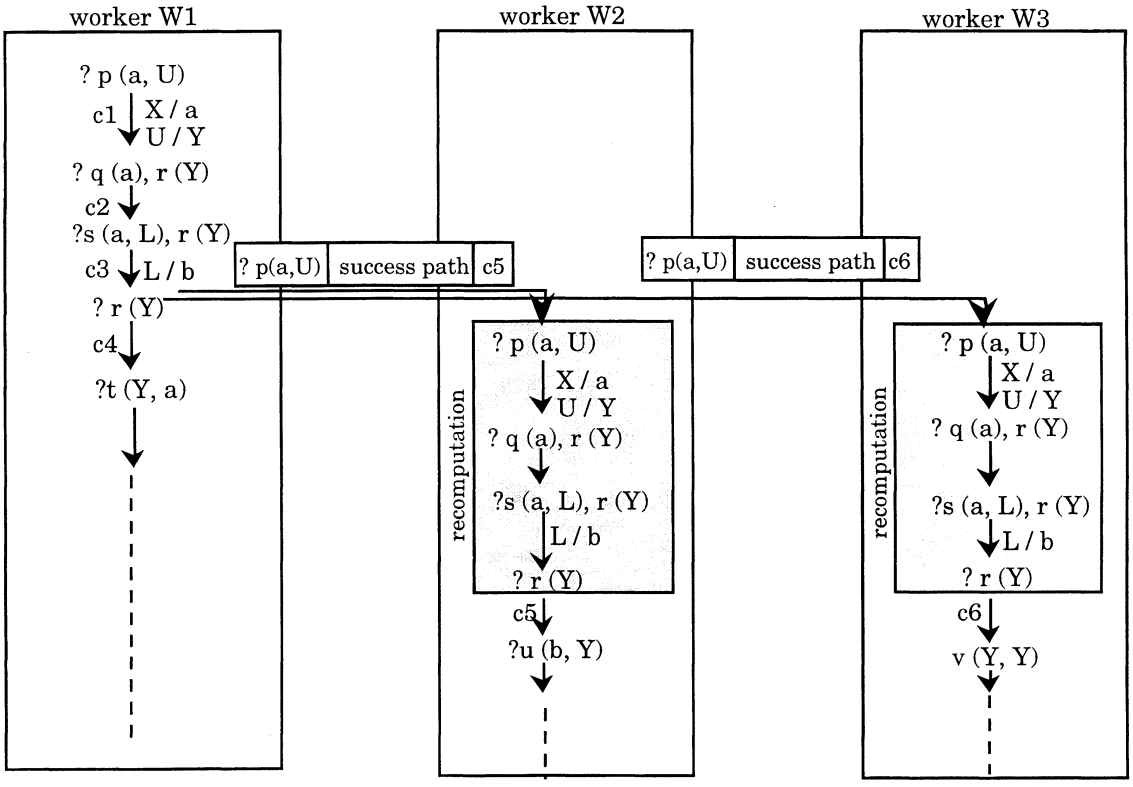


Fig. 5. Disjunctive parallelism exploitation.

```
c1:  p :- q, ...
c2:  *q : - ( r1 & r2 & r3 ).
c3:  *q :- ( s1 & s2 ).
...
```

Fig. 6. Program with conjunctive under disjunctive parallelism.

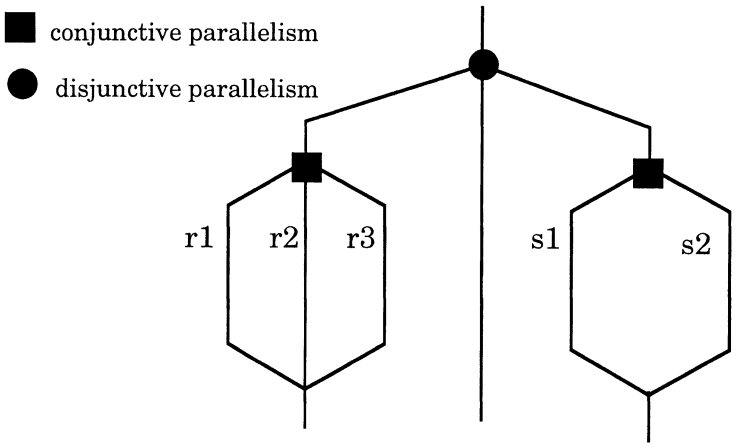


Fig. 7. Conjunctive under disjunctive parallelism exploitation.

2.2.2. Disjunctive parallelism

Disjunctive parallelism is exploited by means of the following multisequential approach: a new sequential environment is generated for each alternative clause by recomputing [2] the initial goal. In this way variables bindings of an disjunctive task are arranged to be entirely independent from their parent and sibling tasks. The recomputation is carried out without backtracking, following the success path (sequence of clauses that have succeeded) recorded by the parent worker. Fig. 5 depicts the control flow of the program of Fig. 4. Two out of three alternatives of the goal $r(Y)$ are scheduled on workers $W2$ and $W3$, and worker $W1$ sends to them the goal, the success path, and the alternative clause to build its environment. The overhead introduced by the recomputation is small provided disjunctive parallelism is exploited in the upper levels of the search tree. This happens when the user annotates disjunctive parallelism with enough granularity.

2.2.3. Conjunctive under disjunctive parallelism

Taking into account that disjunctive parallelism is exploited by creating an independent sequential task for each disjunctive clause, conjunctive under disjunctive parallelism is exploited as multiple conjunctive tasks, as it is shown in the search tree of Fig. 7 for the program in Fig. 6.

2.2.4. Disjunctive under conjunctive parallelism

The PDP approach to exploit disjunctive under conjunctive parallelism [4] avoids storing partial solutions and synchronising workers. This kind of parallelism is exploited by recomputing the execution of the program in such a way that a new combination of solutions is obtained. This procedure is equivalent to a transformation of the program,

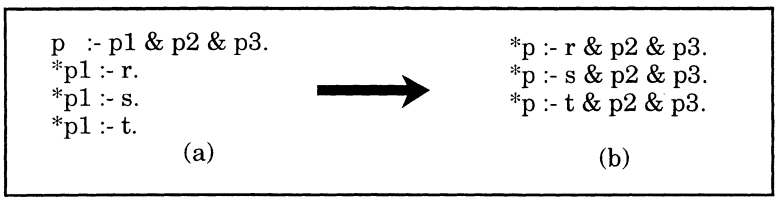


Fig. 8. Program with disjunctive under conjunctive parallelism (a) and equivalent transformation (b).

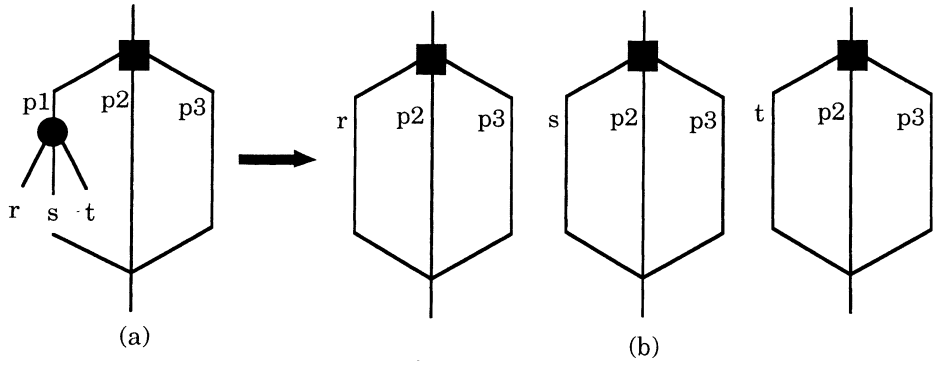


Fig. 9. Disjunctive under conjunctive parallelism exploitation.

```
check(X,Y,Xs,Ys) :- p(X,Xs) & q(Y,Ys).
*p(X,Xs) :- p1(X,Xs).
*p(X,Xs) :- p2(X,Xs).
*q(Y,Ys) :- q1(Y, Ys).
*q(Y,Ys) :- q2(Y, Ys).
p1(X,Y) :- ...
p2(X,Y) :- ...
q1(X,Y) :- ...
q2(X,Y) :- ...
```

Fig. 10. Program with disjunctive under conjunctive parallelism.

as it is demonstrated in the example of Fig. 8. In this way, the different solutions for a goal in a parallel call are not reached by backtracking but by recomputing the parallel call, as shown in Fig. 9. In this way, the PDP method transforms disjunctive under conjunctive parallelism to conjunctive under disjunctive parallelism. This scheme is applied recursively: 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.

In order to illustrate the parallel tasks generated by exploiting disjunctive under conjunctive parallelism let us consider the program in Fig. 10 with combined parallelism:

The *execution time* of the sequential and parallel execution can be represented as it is shown in Fig. 11.

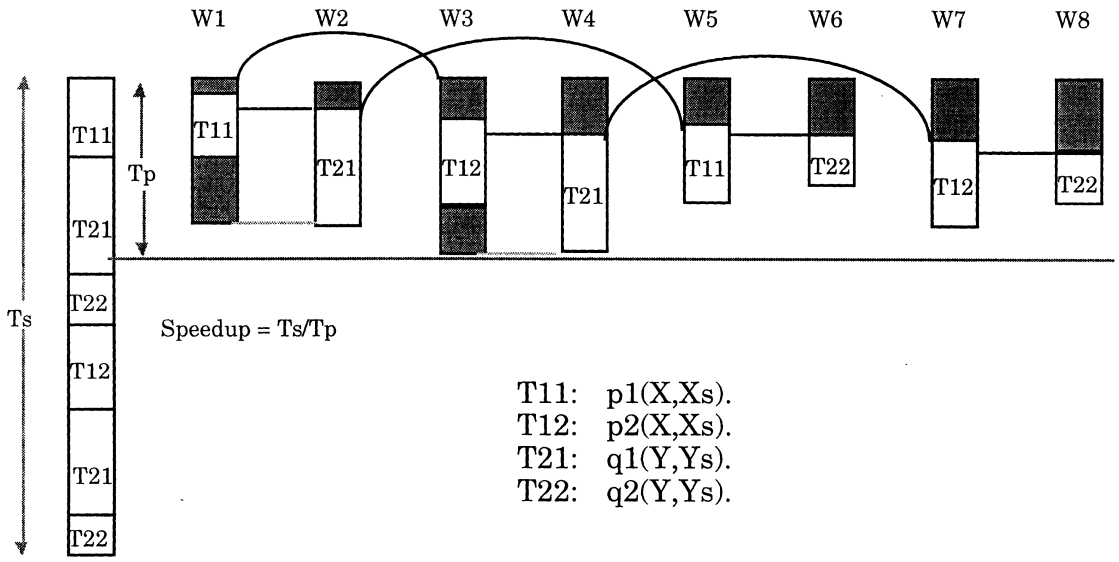


Fig. 11. Parallel execution.

Because of the exploitation of conjunctive parallelism, the tasks $T11$ and $T21$ are executed simultaneously by the workers $W1$ and $W2$, with an overhead incurred due to the recording of the success path and to the communication. Because of the exploitation of disjunctive parallelism, the task $T11$ creates a new computation, which begins executing the task $T12$. It in turn starts the task $T21$ because of the appearance of conjunctive parallelism. Similarly, the execution of the task $T21$ in $W2$ and $W4$ starts new computations in $W5$ and $W7$ (*disjunctive parallelism*), which in turn start the execution of $T22$ in $W6$ and $W8$ because of the exploitation of *conjunctive parallelism*. Since the execution of the task $T21$ takes longer than the execution of the task $T12$, the parent worker $W3$ has to wait for its answer.

2.3. *Functions of the workers*

Each worker implements an extension of the WAM [23]. New data structures and instructions have been added to manage parallelism, mainly for the implementation of the following functions:

- *Success path recording*. 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 who receive the task of exploiting *disjunctive parallelism* are able to follow the success path in order to reach the corresponding choice point without backtracking. The recomputation is optimised by taking advantage of the common information between the tasks successively assigned to a worker, since only the different part needs to be recomputed.
- *Fork and join control of a parallel call*. The parallel execution of a parallel call requires the synchronisation of the reception of the answers. In order to synchronise the execution of a parallel call, we have adopted an extension for a distributed memory architecture of Hermenegildo's system [17] for shared memory architectures.
- *Computation mode*. A worker can operate in different modes depending on the exploited parallelism.

3. 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 must be scheduled for sequential or parallel execution depending on its granularity [11,12,22], i.e. a time estimation of its execution.

Granularity analysis can be performed at compile time, at run time or both. Compile time methods analyse the structure of the program in a static way. However, the cost of most tasks are unknown until parameters are instantiated at execution time, and thus, the result of this analysis may be inaccurate. On contrary, the run time estimation can be more accurate, but at the expense of the introducing such an overhead that the parallel execution is penalised. A good solution consists in obtaining as much information on the cost as possible at compile time, and then using it to complete the cost estimation at runtime with low overhead. This approach has been successfully essayed in some systems [12,22].

PDP, following the latter approach, applies a granularity control for *conjunctive parallelism* exploitation based on the estimation provided by the CASLOG system [12]. However, for *disjunctive parallelism* it uses heuristic observations concerning the level in the search tree in which solutions appear. In PDP, the schedulers distribute parallel tasks (conjunctive and disjunctive) among idle workers, though it is the workers who decide if a task has enough granularity to be executed in parallel.

3.1. Granularity control in disjunctive parallelism

PDP performs a granularity control for the exploitation of disjunctive parallelism based on a heuristic observation: the disjunctive solutions of a goal in the search tree appear at very close levels. When a worker obtains a solution, it records the level reached in the search tree. The pending *disjunctive tasks* are associated to a *choice* point in the search tree. A pending *disjunctive task* is sent to an idle worker only if the distance between the choice point and the level of the last solution is greater than a threshold value (*cost_par*), which depends on the scheduling time of the system, and is tuned experimentally. This test does not introduce run time overhead since only a comparison is required.

The code generated for PDP to introduce this control will be of the form:

$$\text{IF}(ls - li) < \text{cost\_par THEN sequential}(ni) \text{ ELSE parallel}(ni),$$

where ls is the level of the last solution and li the level corresponding to the node ni, whose parallel execution is being decided. Fig. 12 shows two disjunctive nodes, $n1$ and $n2$, and the last solution ns. According to the previous criterion, the parallel tasks arising from $n2$ will be executed sequentially, while those arising from $n1$ will be scheduled in parallel.

The speedup achieved by performing this procedure is shown in Fig. 13 for *query, zebra, master mind* (mm) and *queens* benchmarks. *Query* is a database programs; *zebra* is a puzzle; *master mind* and *queens* are games. For eight workers the speedup is greater for programs with a lot of parallel tasks (*queen10*). The greatest effect of the control is achieved on the system with 15 workers.

3.2. Granularity control in conjunctive parallelism

In the case of conjunctive parallelism, PDP applies a control mechanism based on the estimation provided by CASLOG [12]. This system analyses the worst case cost of a logic program and provides at compile time a granularity estimation of a predicate in the form of an algebraic expression of the size of the input. This expression is evaluated at execution time.

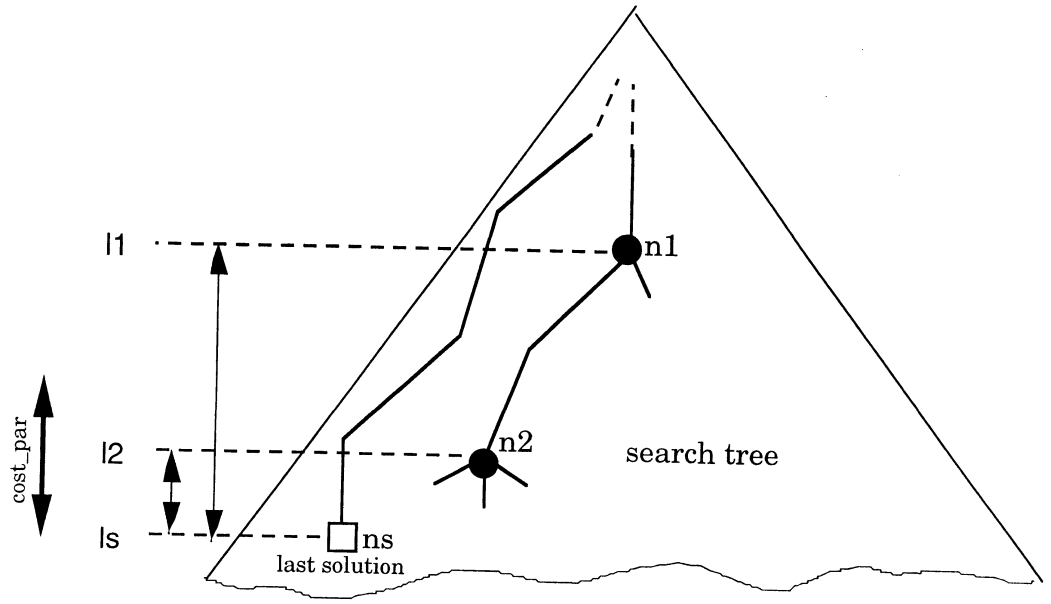


Fig. 12. Granularity estimation for disjunctive parallelism.

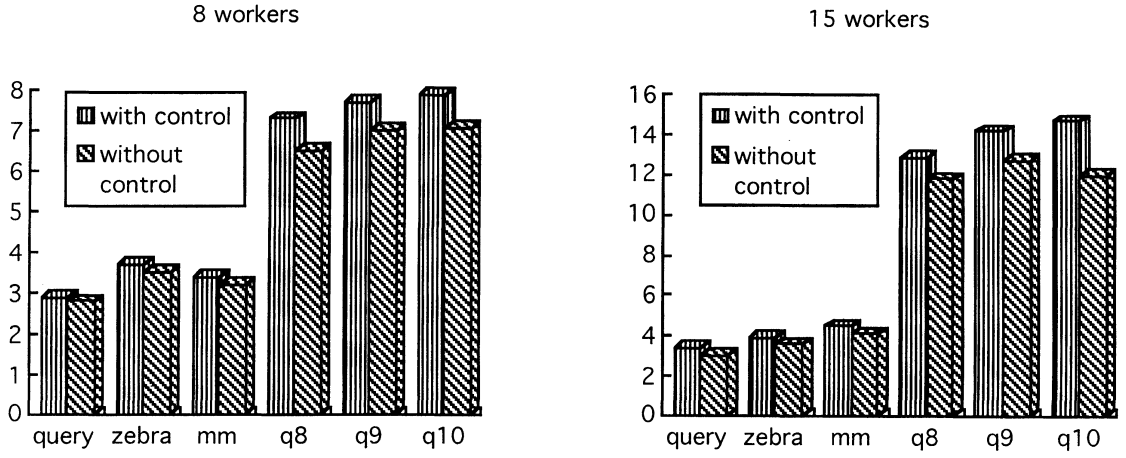


Fig. 13. Speedup with disjunctive granularity control.

Let $p(X, Y)$: -$q(X)\&r(Y)$ be a clause with two independent goals, so that they are candidates for parallel execution, and let $Tq(n)$ be the expression of the granularity estimation provided by CASLOG for an input of size n for the goal $q(X)$. Suppose the cost of creating a parallel task in PDP is *cost_par*. Then the code generated for the clause will be of the form:

$n := size(X)$;
IF $T_p(n) <$ cost_par THEN sequential (q, r) ELSE parallel (q, r).

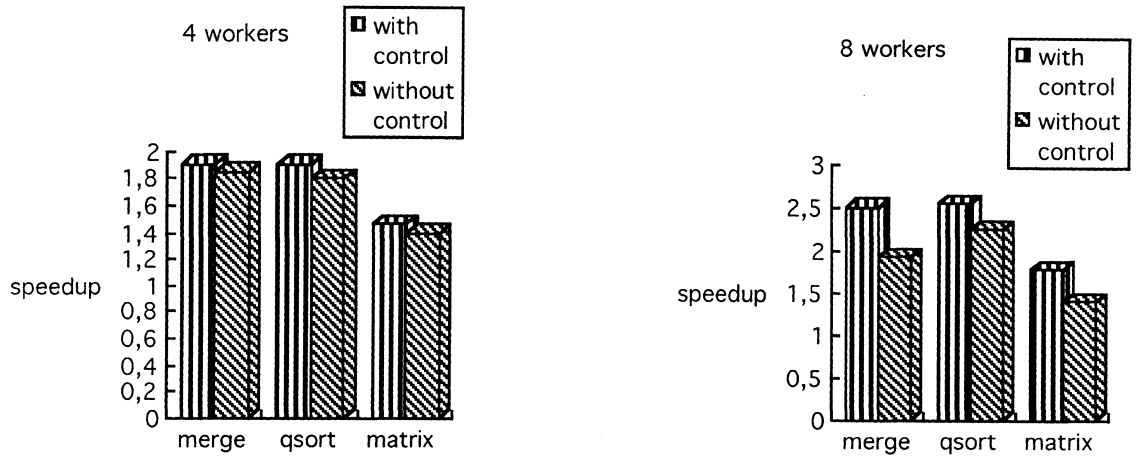


Fig. 14. Speedup with conjunctive granularity control.

That is, if the estimated granularity is greater than the scheduling cost, goals $q(X)$ and $r(X)$ are executed in parallel, otherwise they are executed sequentially.

For example, for the predicate

```
:-mode(flatten/2, [+, −]).
flatten(X, [X]):-atomic(X), X\ == [], !.
flatten([], []).
flatten([X|Xs], Ys):-(flatten(X, Ys1)&flatten(Xs, Ys2)), append(Ys1, Ys2, Ys).
```

the granularity expression is:

$$T_{\text{flatten}}(n) = 0.5n^2 + n + 0.5,$$

where n is the size of the input argument (first argument). The average cost for creating a task in PDP is 30. So, if the size of the input list is 5, the granularity estimation will be $T_{\text{flatten}}(5) = 18 < 30$, and the goals will be executed sequentially. However, if the input list has 12 elements, $T_{\text{flatten}}(12) = 84.5 > 30$, and the goals will be executed in parallel. In this case, the threshold value is 7, that is, only input lists larger than 7 will be executed in parallel.

The overhead introduced by the evaluation of $Tp(n)$ at execution time comes mainly from the evaluation of size (X). However, in the process of creating a parallel task, a PDP worker constructs a message containing the parallel goal and its arguments, and so, the evaluation of the sizes of the arguments introduces almost no overhead in the system.

Fig. 14 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 four workers the control does not have any effect, since the size of the system limits the exploitation of parallelism. With eight workers and the programs *merge* and *qsort*, the control produces the greatest effect. 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.

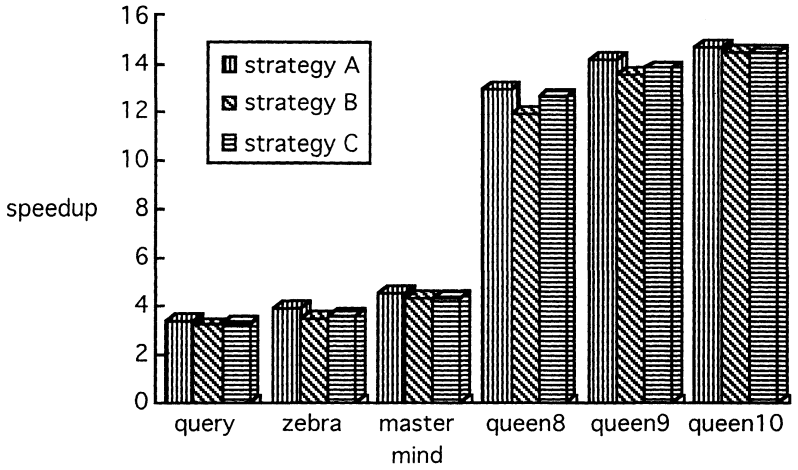


Fig. 15. Speedup for different scheduling policies.

4. Scheduling policy

The scheduling of pending tasks among the workers of the system is an important issue for the overall performance. In distributed memory systems, the information needed to carry out the scheduling is usually centralised, so as to reduce the exchange of messages. PDP has been designed with a hierarchic scheduling policy, in which the schedulers are responsible for the distribution of pending tasks among idle workers.

The hierarchic scheduling policy can be outlined as follows: A worker with pending tasks (of enough granularity) reports it to the scheduler, including the kind of parallelism. Schedulers are notified both, offering and idle workers. When a scheduler receives a warning from an idle worker, it chooses a pending task for it according to the selected scheduling strategy. If a scheduler finds out that every worker within its group is busy, and there are pending tasks in the group, such a circumstance is reported to the scheduler of the upper level. In the same way, if a scheduler finds out that the percentage of idle workers in its group exceeds an experimentally fixed value, then it reports to its upper level scheduler that they will accept tasks.

A worker may be in one out of the three states: *idle* (without work), *working* or *offering* (with pending work). A given scheduler knows the state of the workers in its own cluster. The scheduling policy determines which offering worker has to be requested by which idle worker. To optimise 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 give tasks to an idle worker, the scheduler can follow several strategies previously used by different systems [19,21]:

- *Strategy A*: Choose the nearest offering worker in the cluster.
- *Strategy B*: Choose the most loaded offering worker in the cluster.
- *Strategy C*: Choose the oldest offer in the cluster.

The chart in Fig. 15 shows the speedup for each strategy on a system with 15 workers. Since the measured times differ from run to run, all speedups given are computed using the average times of three runs. The example programs examined are standard benchmarks for *disjunctive parallel systems*: the *queen* problem, *query* (a database problem), *zebra* (a puzzle), and the *mastermind* program.

The results show that the best strategy is A, i.e. choosing the nearest idle worker to the offering worker. This strategy optimises the traffic in the network and favours exchanges between workers which have shared tasks previously,

thus optimising disjunctive parallelism exploitation. The worst strategy is B, since it is the most expensive one (it requires more information exchange than the others), while strategy C (choosing the oldest request) is almost as good as A, since it is the cheapest one requiring no analysis by the scheduler. Nevertheless, the results make evident that the system is almost no sensitive to the different strategies.

5. Implementation on a transputer network

Our current implementation has been made on a Supernode (Parsys) with 16 T800 transputers connected in a torus network. There is a transputer devoted to the input/output functions (IO) directly connected to the host. This PDP implementation has only a cluster composed of a scheduler and 15 workers. The scheduler is placed on a transputer next to the input/output one, in order to receive and send the messages of the user. The remaining transputers support PDP workers which implement an extension of the WAM with new data structures and instructions to manage parallelism. Each PDP instruction is coded in Parallel ANSI C.

In each processor the computation and communication functions have been split. There are three processes controlling the input, output, and computation respectively, as Fig. 16 shows.

The input process awaits (*ProcAltList instruction*) the arrival of messages from any of the input channels 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, which is then taken from the *int_messages* area to be analysed. When a message has to be sent to another processor, a further message is sent through an internal channel to awake the output process.

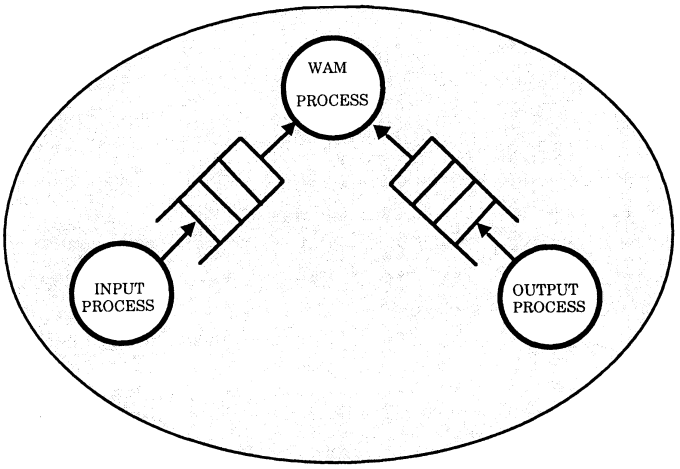


Fig. 16. WAM extended processes.

```
while(1)
    {if thereis_message
        treat_message();
    execute_instruction(opcode);}
```

The output process waits to be awaked (ProcAlt instruction) when a message has to be sent to another processor. The output channel is determined by the *routing* procedure depending on the destination address.

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

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.

6. Performance results

Some sequential programs have been run in order to evaluate the overhead due to the parallel mechanism. The experiments show that this overhead is less than 10%, 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 writing down of the success path is less than 5%. For each kind of parallelism we have investigated the achieved speedup. A worker's time is mainly distributed among the following activities:

- *Execution*. Time spent in executing its sequential tasks.
- *Inactivity*. Time in which the worker is idle, waiting for a new task.
- *Recomputation*. Time spent in recomputing the success path of a new task; this time only appears during the exploitation of *disjunctive parallelism*.
- *Waiting*. Inactivity time due to waiting for the answer from other workers; this time only appears during the exploitation of *conjunctive parallelism*.
- *Communications*. Time spent in communications with the scheduler and other workers; this time includes the time spent in constructing the messages.

6.1. Evaluation of disjunctive parallelism

Fig. 17 shows the speedup achieved by exploiting *disjunctive parallelism* in some benchmarks. All benchmarks have been executed in a "program, fail" way, in order to reach every solution. The speedup is linear for programs with coarse grain parallelism, such as *queen*, and decreases when the granularity of the program becomes smaller (*zebra*).

6.2. Evaluation of conjunctive parallelism

The exploitation of *conjunctive parallelism* in PDP requires high granularity programs, since the workers sharing a task exchange more messages than in the case of *disjunctive parallelism*. Fig. 18 shows the speedup achieved for the programs *qsort, merge* and *matrix*. For programs with more fine grain parallelism, no speedup is achieved by

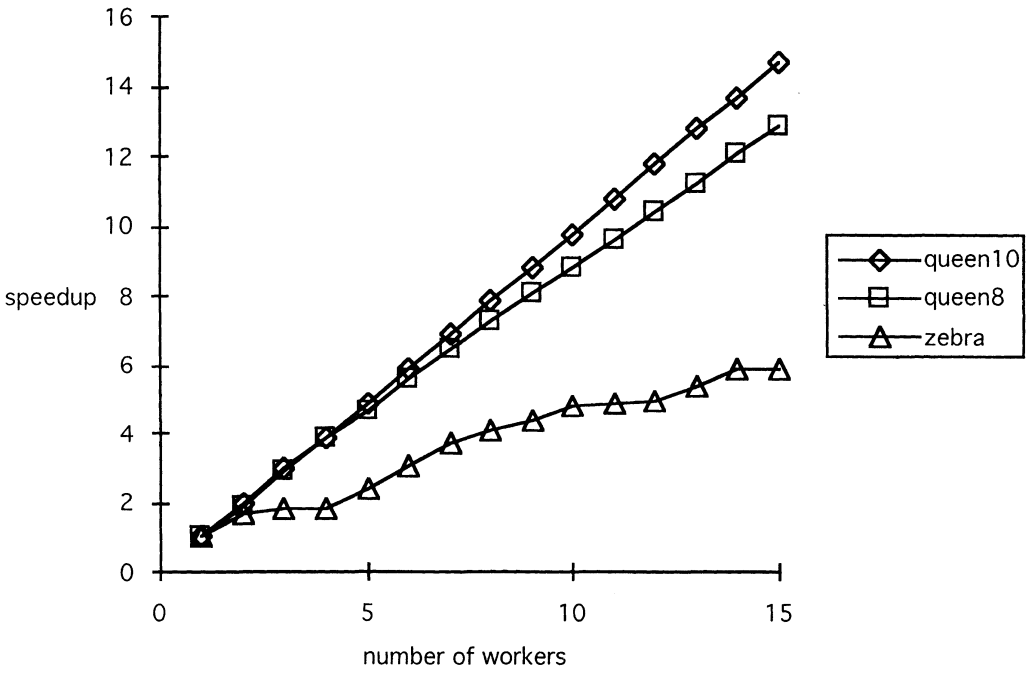


Fig. 17. Speedup for disjunctive parallelism.

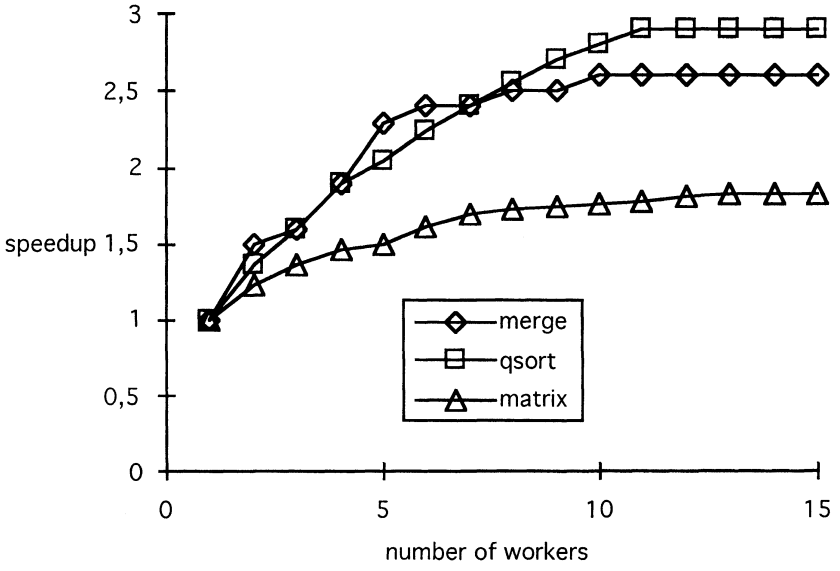


Fig. 18. Speedup for conjunctive parallelism.

```
? 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).
```

Fig. 19. Synthetic 1 benchmark.

```
? 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).
```

Fig. 20. Synthetic 2 benchmark.

exploiting *conjunctive parallelism*. Results show that the amount of parallelism exploited is small and the execution time does not decrease from 10 workers upwards.

6.3. Evaluation of combined parallelism

In order to evaluate the behaviour of PDP for programs with both kinds of parallelism, synthetic benchmarks with coarse grain parallelism have been run. The first one, *synthetic 1*, shown in Fig. 19, presents *conjunctive under disjunctive parallelism*. There is disjunctive parallelism in the procedure *check*, while *conjunctive parallelism* appears in the body clauses of this procedure.

The benchmark *synthetic 2*, shown in Fig. 20, presents *disjunctive under conjunctive parallelism*. There is *conjunctive parallelism* in the body of the *check* clause, while the procedure *p* presents *disjunctive parallelism*.

Figs. 21 and 22 present the speedup achieved by exploiting each kind of parallelism. Results show a significant speedup for all executions exploiting parallelism. The benchmark 1 has been chosen to show the advantages of exploring different solutions at the same time, since the first solution explored by the sequential machine can be slower to reach (the second clause of *check* is computed in a shorter time than the first one). It can be observed that when both kinds of parallelism are exploited, the performance improves in all cases. The speedup achieved when exploiting both kinds of parallelism is greater than the product of the speedups achieved by exploiting each kind of parallelism separately. The reason for this is that the exploration of the different solutions when *conjunctive parallelism* is exploited requires a number of message exchanges between the parent worker and the worker exploring each goal in the parallel call (new solution requests and answers) that are avoided when *disjunctive under conjunctive parallelism* is exploited.

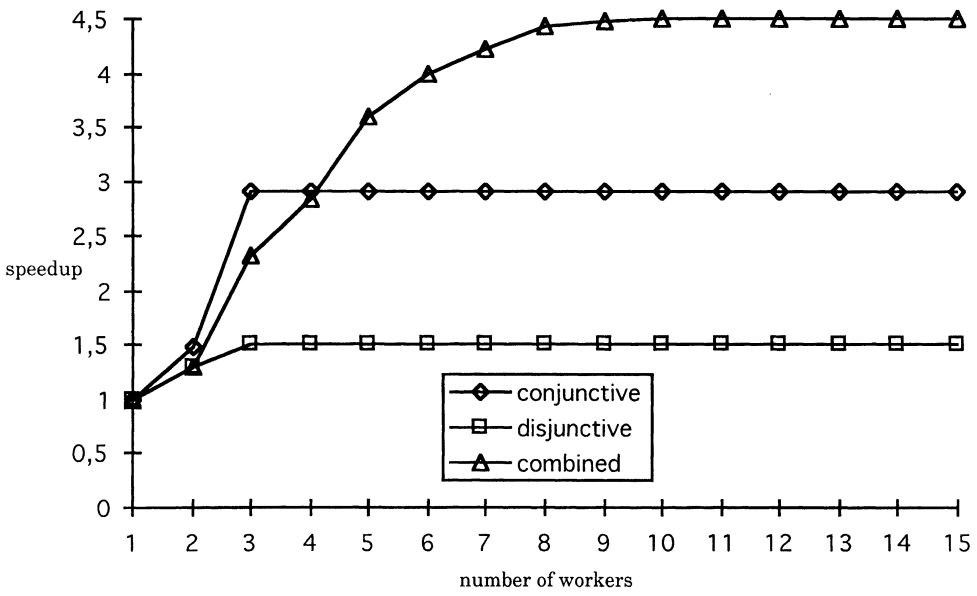


Fig. 21. Speedup for synthetic 1.

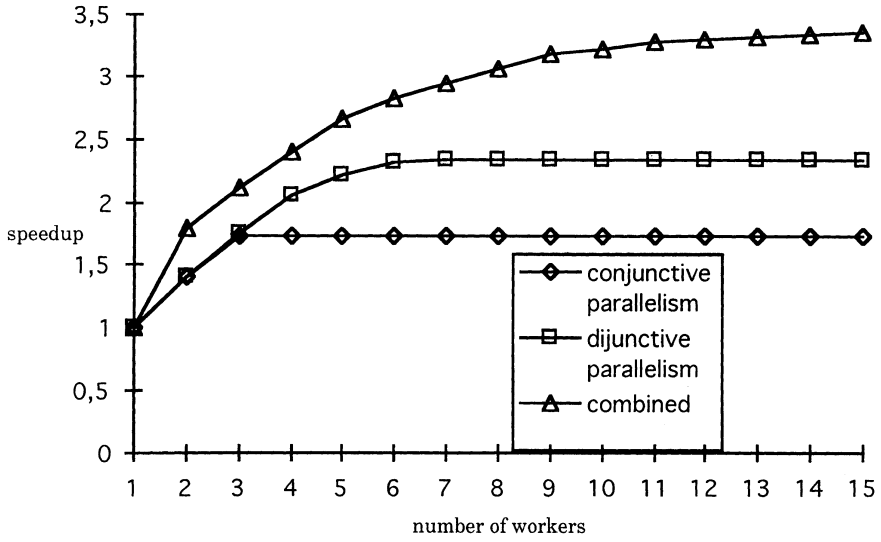


Fig. 22. Speedup for synthetic 2.

6.4. Absolute execution time

The design of PDP is aimed to improving the execution speed of Prolog programs on a distributed memory system following a multisequential approach. The sequential component could be improved either by using faster processors or by introducing more optimisations. Therefore, the absolute times are less significant than the speedup.

Table 1
Absolute execution times in ms for disjunctive parallelism

Program	One worker	Three workers	Eight workers	15 Workers
zebra	1160	644	400	293
queen8	39 040	13 943	5348	3026
queen10	1 028 119	331 651	130 142	69 940

Table 2
Absolute execution times in ms for conjunctive parallelism

Program	One worker	Three workers	Eight workers	15 Workers
merge	830	360	332	307
qsort	1289	560	460	379
matrix	1394	996	820	774

Table 3
Absolute execution times in ms for the three kinds of parallelism

Program	Sequential	Disjunctive	Conjunctive	Combined
Synthetic 1	740	493	255	164
Synthetic 2	9108	3843	7784	1989

Nevertheless, we present in Tables 1–3 the absolute times for the benchmarks executed on Supernode with T800 transputers.

7. Conclusions

The development of PDP has achieved some very interesting and important results for logic programming systems. Disjunctive parallelism exploitation provides a linear speedup for high granularity programs. The recomputation approach has been shown to give a high performance, further improved with increases in system size. The exploitation of conjunctive parallelism also provides a significant speedup for coarse grain programs, though 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 for this is that the exploration of the different solutions when conjunctive parallelism is exploited requires a 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 avoided when combined parallelism is exploited.

Granularity controls have also been introduced for each kind of parallelism. For conjunctive parallelism PDP applies a control based on the estimation provided by the CASLOG system, which gives an upper bound to the cost of a large class of logic programs. This control has been successfully applied to some shared-memory parallel systems, but it is particularly suitable for PDP because it can be applied without the introduction of additional overhead. For disjunctive parallelism PDP controls granularity by applying a heuristic-based method, which can be adapted to other parallel systems. The introduction of these controls has been proved to be necessary to avoid the deterioration of the performance when the system size increases. Different scheduling policies have been tested. The best results have been achieved when the nearest idle worker was chosen.

We are now porting the PDP system to another distributed memory platform consisting of a workstation network. This will allow us to extend the system to a larger number of workers and to try more applications. It would also be worthwhile to investigate the construction of a mixed system, with some shared memory used for the exploitation

of pure conjunctive parallelism, devoting the distributed memory to the exploitation of disjunctive parallelism and combined parallelism.

Acknowledgements

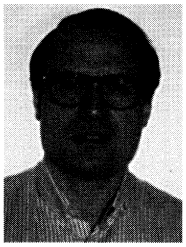
The authors would like to express their gratitude to the referees for their detailed revisions and helpful comments.

References

[1] K.A. Ali, R. Karlsson, The muse approach to or-parallel prolog, Int. J. Parallel Program. 19 (2) (1990) 129–162.
[2] L. Araujo, J.J. Ruz, OR-parallel execution of prolog on a transputer-based system, in: Transputers and Occam Research: New Directions, IOS Press, 1993, pp. 167–181.
[3] L. Araujo, J.J. Ruz, A transputer-based prolog distributed processor, Proceedings of the 17th World Occam and Transputer User Group Technical Meeting (WoTUG-17), Bristol, UK, 1994, pp. 205–219.
[4] L. Araujo, J.J. Ruz, PDP: Prolog distributed processor for independent OR parallel execution of Prolog, Proceedings of the International Conference on Logic Programming, MIT Press, Cambridge, 1994, pp. 142–156.
[5] D. Arnold, Z. Farkas, G. Gerlei, K. Molnar, G. Umann, ZEXPERT: A Prolog-based expert system shell, Proceedings of the Practical Application of Prolog, London, 1992.
[6] J. Bellone, A. Chamard, C. Pradelles, PLANE: An evolutive planning system for aircraft production written in CHIP, Proceedings of the Practical Application of Prolog, London, 1992.
[7] P. Biswas, S. Su, D. Yun, A scalable abstract machine model to support limited-OR(LOR)/restricted-AND parallelism(RAP) in Logic Programs, Proceedings of the International Conference on Logic Programming , MIT Press, Cambridge, MA, 1988, pp. 1160–1179.
[8] A. Calderwood, P. Szeredi, Scheduling Or-parallelism in Aurora – the Manchester scheduler, Proceedings of the International Conference on Logic Programming, MIT Press, Cambridge, MA, 1989, pp. 419–435.
[9] J.H. Chang, A. Despain, D. Degroot, AND-parallelism of logic programs based on a static dependency analysis, Proceedings of Spring Compton, IEEE Press, New York, 1985, pp. 218–225.
[10] J.S. Conery, Binding environments for parallel logic programs in non-shared memory multiprocessors, Int. J. Parallel Program. 17 (2) (1988) 125–152.
[11] S.K. Debray, N.-W. Lin, H. Hermenegildo, Task granularity analysis, SIGPLAN-90 Conference on Programming Language Design and Implementation, ACM, New York, 1990, pp. 174–188.
[12] S.K. Debray, N. Lin, Automatic complexity analysis of logic programs, Proceedings of the International Conference on Logic Programming , MIT Press, Cambridge, MA, 1991, pp. 599–613.
[13] D. Degroot, Restricted AND-parallelism, Proceedings of the International Conference Fifth Generation Computer Systems , ICOT, 1994, pp. 471–478.
[14] A. Despain, The design system (ASP) of the Aquarius Project, Proceedings of the Second International Workshop on VLSI Design, December 1988.
[15] A. Despain et al., Aquarius, Computer Architecture News, March 1987.
[16] G. Gupta, M. Hermenegildo, V.S. Costa, And-Or parallel Prolog: A recomputation based approach, New Generation Computing 11 (3–4) (1993) 297–322.
[17] M. Hermenegildo, An abstract machine based execution model for computer architecture, design and efficient implementation of logic program in parallel, Ph.D. thesis, University of Texas, Austin, 1986.
[18] M. Hermenegildo, K. Greene, The system: Exploting Independent And-Parallelism, New Generation Computing 9 (3–4) (1991) 233–257.
[19] H. Kuchen, A. Wagener, Comparison of dynamic load balancing strategies, Technical Report 90-5, Aachener Informatik-Berichte, 1990.
[20] K. Muthukumar, M. Hermenegildo, Determination of variable dependence information at compile-time through abstract interpretation, Proceedings of the North American Conference on Logic programming, MIT Press, Cambridge, MA, 1989, pp. 166–185.
[21] M. Sugie, M. Yoneyama, T. Tarui, Load-dispatching strategy on parallel inference machine, Procedings of the International Conference on Fifth Generation Computer Systems, 1988, pp. 987–993.
[22] E. Tick, Compile-time granularity analysis for parallel logic programming languages, New Generation Computing 7 (1990) 325–337.
[23] D.H.D. Warren, An abstract prolog instruction set, Technical Report 309, SRI International, 1983.
[24] D.H.D. Warren, Or-parallel execution models of Prolog, Proceedings of the International Joint Conference on Theory and Practice of Software Development, Springer, Berlin, 1987, pp. 243–259.

[25] H. Westphal, P. Robert, J. Chassin, J. Syre, The PEPsys model: Combining backtracking, AND- and OR-parallelism, Proceedings of the International Logic Programming Symposium, 1987, pp. 436–448.

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



Jose J. Ruz received the B.S. degree in Physics from the Complutense University of Madrid in 1974 and the Ph.D. in Computer Science in 1980 from the same University. He is currently an Associate Professor in the Department of Computer Science at the Complutense University of Madrid, Spain. In the past he has worked on concurrent database machines. His current research interests include parallel computer architectures and all aspects of parallel execution of declarative languages with emphasis on constraint programming. He also is a member of the IEEE Computer Society.