ELSEVIER

Journal of Systems Architecture 43 (1997) 437–457

JOURNAL OF SYSTEMS ARCHITECTURE

Massively parallel execution of logic programs: A static approach

F. Baiardi [a,*], A. Candelieri [b], L. Ricci [a]

[a] *Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy*
[b] *Ist. per la Ricerca sui Sistemi Informatici Paralleli, CNR, Napoli, Italy*

Abstract

A static model for the parallel execution of logic programs on MIMD distributed memory systems is presented where a refutation is implemented through a process network returned by the compilation of the logic program. The model supports Restricted-AND, OR and stream parallelism and it is integrated with a set of static analyses to optimise the process network. Altogether, the processes interact according to a static data driven model with medium grain operators. Data flowing in the network is tagged to distinguish bindings belonging to the same refutation. A scheduling strategy to integrate low level scheduling and message flow control has been defined. Performance figures are presented.

Keywords: Logic language; Distributed memory system; Compilation; Static process network; Congestion

1. Introduction

The parallel implementation of logic languages is currently pursued as a cost effective solution to speed up the execution of logic programs. Most of the current approaches focus on the definition of dynamic process networks, where several instances of an interpreter are created when the search reaches a branch point of the search tree [9,10,15,17–19,21–23,25,26,32–34,36,37,39]. This approach has been originally developed in the case of shared memory systems because it is simplifies load balancing. Current work is focused on the implementation of this approach on distributed memory systems. Distributed memory systems are becoming increasingly popular because of their scalability and cost effectiveness. These architectures include a large amount of processing elements, PEs, each including a processor and some local memory. Either a direct or an indirect partial interconnection network supports the cooperation among PEs.

In a distributed memory system, the effectiveness of a dynamic approach can be largely reduced because of the overheads due to the partial interconnection network. As an example the complexity of process mapping increases due to *the lack of infor-*

* Corresponding author. Email: baiardi@dipisa.pi.unipi.it.

PII S1383-7621(96)00056-2

mation about process communication. It may be the case that two heavily communicating processes, P and Q, are not mapped onto directly connected PEs because the mapping algorithm cannot anticipate the creation of Q when P is mapped. This results in an overhead due either to the routing of messages between P and Q or to the migration of P and/or Q to two directly connected PEs. Further overheads may arise because of the transmission of the process data and/or code to a PE where a process is mapped [24,29].

A static approach [18–20,24,32], implements a refutation through a concurrent program where the number of processes and the interconnection among processes are *static* and are produced by the compiler. The compilation of a program LP and of a query Q consists of two steps. The first step returns Net(Q,LP), the network of processes implementing the refutation of G through the program LP. Information to optimise Net(Q,LP) is deduced through a set of static analyses of LP.

The second step of the compilation maps the processes of Net(Q,LP) onto the PEs [2,5,18,24].

Since the structure of Net(Q,LP) is known, the mapping step can exploit any information about process communications to improve the mapping. As a counterpart, this structure is independent of the target architecture and, to map Net(Q,LP) onto the architecture, sharing of some resources may be necessary. A proper scheduling strategy has to solve the conflicts arising because several processes share a PE or several communications share a physical link.

Obviously, a static approach can fail to exploit those opportunities for parallelism that depend upon the values of some variables. The number of lost opportunities can be reduced by a proper set of static program analysis.

This work introduces a static concurrent execution model where a set of static analyses of the logic program determines the structure of the concurrent program to implement a refutation. The analyses do not require any information about the usage of the various predicates. Their correctness has been proved through an abstract interpretation approach [1,8,12, 17,19,20,27,28,32].

To prevent both congestion and hot spots [14,15,31], a strategy to schedule both the processors and the communication links has been defined that bounds the number of messages flowing in the interconnection network simultaneously.

The paper is organised as follows. Section 2 introduces the static execution model and it points out where the output process network can be optimised according to the information returned by the program static analyses. Section 3 briefly defines the basic notions underlying the analyses and the analyses themselves. Section 4 discusses the dynamic scheduling strategy and an implementation of the model. Performance figures are discussed.

We do not discuss mapping strategies because, while most mapping strategies may be adopted, experimental results point out that resource scheduling is the most critical issue.

2. The static execution model

We define the concurrent static execution model in the general case of Horn clause logic and refer to [18] for a classification of the various kinds of implicit parallelism of a logic program.

The proposed model implements the refutation of a query $Q(\ldots)$ on a program LP through a process network Net(Q,LP). From now on we drop the dependency of the process network from the logic program, i.e. Net(Q) is used in place of Net(Q,LP).

The compiler produces the network Net(Q), the predicate network for Q, by properly composing the networks Net(Q_1),..., Net(Q_n), *clause networks*, where $Q_1,\ldots,Q_n$ are all and only the program clauses having Q as their predicate symbol.

In turn, each Net(Q_i) is recursively defined in terms of the networks INet(L_{i1}),...,INet(L_{im}) where $1 \le j \le m$, L_{ij} is the j-th literal in the body of Q_i

and INet(L_{ij}) is an instance of Net(L_{ij}), the predicate network of L_{ij}. To avoid bottlenecks, Net(Q_i) includes a distinct instance of Net(P) for each literal whose predicate symbol is P.

The strategy to compose Net(Q_1),...,Net(Q_n) to produce Net(Q) defines the solution to OR parallelism, while that to compose INet(L_{i1}),..., INet(L_{im}) to produce Net(Q_i) is a solution to AND parallelism, see Fig. 1.

Each message flowing in the network codifies a goal to be refuted by a network. A message may be described as an Activation Frame, AF, that includes an environment and a store. An environment records a pointer to the store for each variable, while the store records the bindings for the variables. If the refutation of a goal $P(\ldots)$ requires that of the subgoal $R(\ldots)$, AF_R, an AF that codifies this subgoal is sent to I_R, the proper instance of Net(R). The store and the environment of AF_R are subsets of those of AF, they include all and only the bindings of the variables appearing in $R(\ldots)$. In this way, I_R receives the bindings required to refute R only. If the refutation of $R(\ldots)$ is successful, the bindings of

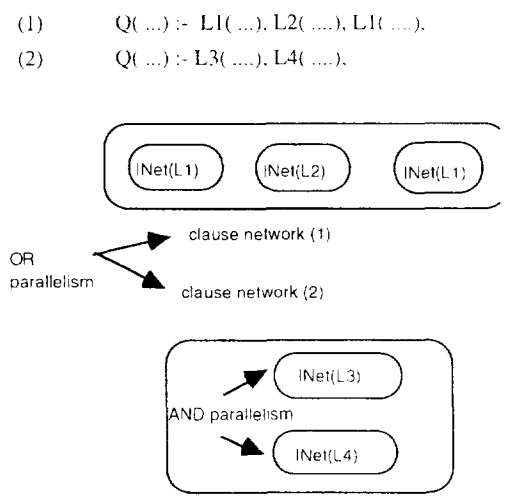

Fig. 1. OR and AND parallelism in the model.

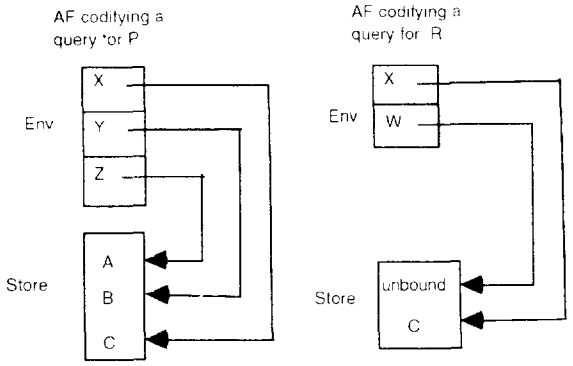

Fig. 2. Creation of the AF to refute $R(X,W)$ in a clause $P(X,Y,Z) :- R(X,W),\ldots,$.

the AF that codifies its results are imported into the AF for the original query $P(\ldots)$, see Fig. 2.

In this way, the processes of Net(Q) cooperate in a data driven way and they exchange AFs that codify the queries for the various subnetworks of Net(Q). Each process implements a predefined set of operations, i.e. the unification of a goal with the head of a clause, the management of recursion and the update of an AF through the bindings returned by the refutation of a subgoal. Further predefined processes are required to implement non-logical predicates of the considered language.

As detailed in the following, a process network implements several refutations in parallel. Consistency is preserved through a tag associated with each AF to distinguish bindings belonging to distinct refutations [3].

2.1. Implementation of OR parallelism

To fully exploit OR parallelism, Net(Q), the predicate network for Q is the parallel composition of all the clause networks Net(Q_1),...,Net(Q_n) where $Q_1,\ldots,Q_n$ are the clauses having Q as their predicate symbol. Net(Q_1),...,Net(Q_n) are composed in parallel through an instance Dist of the predefined

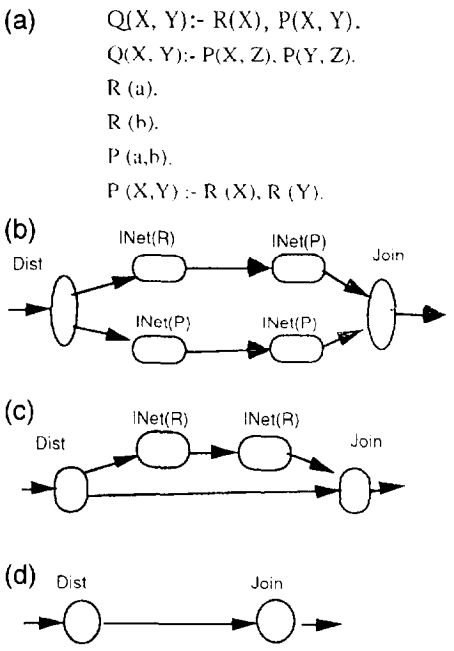

Fig. 3. A program and the corresponding process networks a. A simple program. b. Net(Q). c. Net(P). d. Net(R).

unification process and one instance of the predefined *collector process*, Join, see Fig. 3a and Fig. 3b.

After receiving an AF, Dist unifies the corresponding query with the head of each clause Q_i. If the unification is successful, a frame AF_i is produced and sent to Net(Q_i). The environment of AF_i includes all and only the variables appearing in Q_i. The bindings of the global variables are those returned by the unification, while any local variable is initially unbound. AF_i and AF are then paired with a unique tag, to record that the bindings of the global variables in AF_i returned by the refutation have to be imported into AF.

If the unification of the goal with the head of Q_i fails, no data is sent to Net(Q_i). After attempting the unification with the head of all the clauses, Dist sends to Join(Q) the AF it has received.

To implement a fact, Dist transmits the AF resulting from a successful unification to Join. No backtracking strategy is required because Dist attempts the unification of a goal with all the clauses.

When Join receives an AF_i from Net(Q_i), it imports the binding of AF_i into the corresponding AF received from Dist. The resulting AF is transmitted to the next network. The use of tags makes it possible to correctly and efficiently associate each AF_i with the proper AF.

Because of OR parallelism, any predicate network can produce several AFs, even when receiving a single AF. Hence, both the input and the output of a network are streams of AFs. A predicate network solves distinct queries in parallel, because Dist receives an AF from its input stream as soon as it has unified the previous one against the heads of all the clauses of Q, and sent the resulting AFs to the proper clause networks.

2.2. Implementation of AND parallelism

According to previous experiences [6,9–11,17,18,22,23], two predicate networks implementing literals of the same clause are executed in parallel only if the corresponding literals cannot share an unbound variable, *Restricted-AND parallelism*, RAP. The absence of sharing is deduced through a static share analysis [32], see Section 3.

If a sharing exists, or if the absence of sharing cannot be proven, *stream parallelism* is applied and the two networks are composed according to a pipeline strategy where processes in the first one bind the shared variable and those in the other consume, i.e. check, the binding. While the correctness of the implementation does not depend upon which network is chosen as the producer, the execution time is reduced if the producer network binds the shared variable to a ground term. The producer network is chosen through a static analysis based upon the notion of *ground dependency* [19].

If RAP cannot be exploited, *pipeline-AND parallelism*, an optimisation of stream parallelism may be adopted. This kind of parallelism arises if the consumer network can perform some *useful computation as soon as a partial binding for the shared variable has been produced.* In this case, the producer communicates to the consumer a partial binding for the shared variable, rather than a complete one, as soon as such a binding has been produced.

2.2.1. Restricted-AND parallelism

Within a clause process, the parallel execution of INet(L_{i1}),...,INet(L_{in}) is implemented through two instances of the predefined processes Split and Merge.

After receiving an AF, Split produces an AF_j for each INET(L_{ij}). AF_j includes the bindings in AF of the variables referred to by L_{ij}. After receiving an AF_j from each INET(L_{ij}), Merge produces and sends to the next network an AF including all the bindings of $AF_1,\ldots,AF_n$, see Fig. 4.

To guarantee a consistent merge of the bindings, Split inserts the same tag into all the AFs returned by the decomposition of the same AF.

If $m_j(t)$ denotes the number of AFs with tag t that Merge has received, at a given time, from INET(L_{ij}), $1 \le j \le n$, then, upon receiving an AF with tag t from INET(L_{ik}), $1 \le k \le n$, Merge appends $m_1(t) * \ldots * m_{k-1}(t) * m_{k+1}(t) * \ldots * m_n(t)$

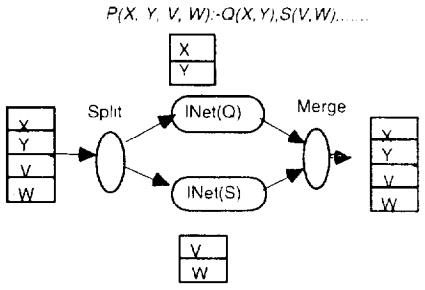

Fig. 4. Composition of two networks to exploit RAP.

AFs to its output stream. Each AF is produced by merging the received AFs with $n-1$ AFs, each received from a distinct subnetwork. This implies that an instance of Merge has to record all the AFs it receives from any INET(L_{ij}). To prevent memory overflows, Merge may be decomposed into a set of parallel processes.

To retrieve the AFs to be merged with a given one while avoiding a sequential scanning, Merge uses the tags paired with the AFs as keys in a hash table.

2.2.2. Pipeline-AND parallelism

This kind of parallelism will be illustrated through an example. Consider the clause:

Transform(X,Y,Z) : −Mol(Y,Y_1), Add(X,Y_1,Z)

and suppose that:

- X, Y, Y_1 and Z are lists, X is ground;
- the refutation of Mol and that of Add returns a ground substitution for an element of Y_1, or of Z, if the corresponding element of Y, or of X and Y_1, is ground.

Let Prod be the subnetwork of Net(Mol) that computes a ground substitution for the first element of Y_1. When Prod generates such a substitution, it sends an AF recording this partial binding both to Net(Add) and to the next subnetwork of Prod in Net(Mol). These two networks can start, respectively, the computation of the first element of Z and that of the second element of Y_1, and so on, see Fig. 5. The correctness of the incremental construction of the bindings is preserved through proper tags in the AFs.

2.3. Networks for recursive clauses

The procedure previously outlined to compile a predicate into a predicate network does not terminate

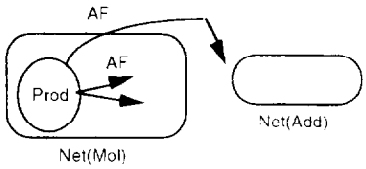

Fig. 5. Interconnection for pipeline-AND parallelism.

if it is applied to a predicate Q such that $Q(\ldots)$ is invoked during the refutation of $Q(\ldots)$.

To avoid this case, we introduce a normal form of logic program characterised by the *absence of mutually recursive predicates*, i.e. no predicate P is defined in terms of a predicate Q $(\neq P)$ that is, in turn, defined in terms of P. *Mutual recursion is removed at compile time* by considering two mutually recursive predicates P and Q as instances of a normal form predicate PQ where a further argument distinguishes the instance, P or Q, to be applied [35], see Fig. 6.

Normal form programs simplify the mapping step because they are compiled into networks where processes are connected in a regular topology [5]. As an example, each network receives (sends) its AFs from (to) one stream only.

The network for a recursive clause includes a distinct instance of RecHan, the predefined process for recursion handling for each recursive invocation. To handle a recursive invocation, RecHan:

(a) records the AF that has produced the invocation,
(b) produces an AF codifying the invocation,
(c) sends to Dist the AF produced in (b),
(d) after receiving from Join the AF codifying the results of the invocation, it builds the proper solutions through the AFs recorded in (a).

P(...) :- R1(....), ..., Q(....), P(....).
P(...) :- ...., P(....), ...
Q(...) :- R2(....), ..., Q(....), P(....).

becomes

PQ(..., p) :- R1(....), ..., PQ(... , q), PQ(.... , p).
PQ(...., p) :- ...., PQ(.... , p), ...
PQ(... , q) :- R2(....), ..., PQ(... , q), PQ(... , p).

Fig. 6. Transformation to remove mutual recursion.

Even if the AF received in (d) is produced by Dist, it is sent to RecHan through Join so that Dist can manage in a uniform way the AFs returned by a unification.

In this way, the process network refutes in parallel goals received either from the previous network or from the RecHan processes within the network itself. In other words, the network input stream merges the output stream of the previous network and the recursive invocations produced by the RecHan processes within the network. Again, consistency is preserved by associating each AF with a unique tag.

If a RecHan receives an AF with tag T, AF(T), and it produces a recursive invocation with tag T_1, then T_1 is a tag depending on T. Upon receiving an AF(T_1) codifying a final substitution, the RecHan produces the results of the invocation by applying these substitutions to AF(T). Because of OR parallelism, a RecHan may receive several AFs, with the same tag. For each AF, either it produces a recursive invocation, with a distinct tag, or it uses the bindings in the AF to produce the result of an invocation.

Dependencies among tags are represented through a tree or a cactus stack, as shown in Fig. 7. A RecHan handles a distinct tree, or a cactus stack, for

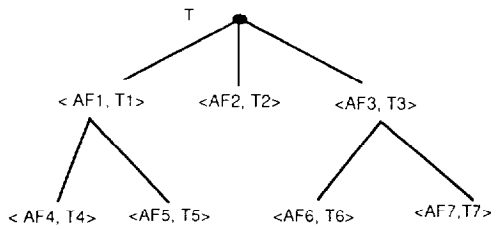

Fig. 7. A tree recording dependencies among tags.

each tag T where T has not been produced by the RecHan itself or, from another point of view, T does not depend upon any other tag.

2.4. Compiling a predicate into a process network

The procedure to compile a predicate P of a normal form program LP into a process network Net(P) may be outlined as follows:

(a) Net(P) is the parallel composition of the clause networks of P through an instance of Dist and one of Join.
(b) A clause network is
1. the sequential composition; or
2. the parallel composition of the networks for the literals in the body of the clause through an instance of Split and one of Merge.

(c) In a clause network of P, the network for a literal $Q(\ldots)$ is
1. an instance, INet(Q), of Net(Q) if $Q \neq P$;
2. a RecHan process if $Q = \mathrm{P}$.

The refutation of a query

$$Q = P_1(\ldots), \ldots, P_n(\ldots),$$

where $P_1, \ldots, P_n$ are predicates of LP, is implemented through a query network, Net(Q), that is the clause network of the clause $P_1(\ldots), \ldots, P_n(\ldots)$.

The input of the query network is an AF codifying the bindings of the variables in the query. The output is a sequence of AFs, each codifying a substitution satisfying the query itself. Since an AF is discarded upon the failure of a unification, no AF is produced by the query network if no such a substitution exists. In other words, *the model backward semantics* [17] *discards an AF* as soon as it is detected that its bindings do not lead to a refutation of the goal. This points out that our model is "all solutions" oriented.

A refutation terminates after producing *all the substitutions for the query.* The termination can be detected through the exchange of further messages, *end messages* [32]. An end message is transmitted to the query network after the AF codifying the query and proper rules are defined so that each network propagates the end messages only if and when it will not produce any further AF.

The proposed static model does not impose an "a priori", architecture dependent, constrain on the degree of parallelism of the output program. The finite amount of PEs is considered in the mapping step and in the resource scheduling strategy

It is worth noticing that a query network includes several instances of distinct processes. Each instance can be further specialised according to the predicate it implements or to its position in the query network. Furthermore, each predefined process can be decomposed in parallel depending upon the optimal process granularity of the target architecture.

3. Static analyses of a logic program

The step of the compilation that generates the process network exploits the information returned by a set of static analyses to optimise the network structure.

The analyses are defined according to a *bottom up approach* [28] so that they return only the information that can be deduced from the structure of the program.

The analyses are based on the notion of *ground dependency*, i.e. the relation between the distinct arguments of a predicate that a successful refutation binds to a ground term. This notion allows the analyses to overcome several limitations of existing analyses. The correctness of the analyses has been formally proved through abstract interpretation techniques. The abstract interpretations have been defined according to a bottom up approach as well [19,20].

```
Select (cons(X , Xs), X, Xs ).
Select (cons(Y, Ys), X, cons(Y, Zs )) :-
Select (Ys, X, Zs).
P (X ,Y, W) :- Select (X, Y, W), F1 (X, Y), F2 (Y, W)
```

Fig. 8.

3.1. Bottom up analyses

First, we point out the main differences between a top down analysis and a bottom up one.

Consider, as an example, the program fragment in Fig. 8, where Select(X,Y,Z) if the list Z is produced by removing one occurrence of Y from X. F1 and F2 are verified if a given relation exists between Y and, respectively, the original list X and the resulting list W.

If the whole program is top-down analysed, some information has to be supplied by the user about the goals that will be refuted and/or the status, i.e. ground or not ground, of some variables in the query. This information is propagated by the analysis till the invocation of P and it defines the context of the invocation. Suppose that, according to the context, the first argument of P is ground, hence the first argument of Select is ground and the other arguments of Select will be ground after the refutation of Select. This information is now included in the contexts of F1 and F2.

A bottom up mode analysis, instead, propagates information starting from the unit clauses of the program. Due to the lack of user supplied information, less information about the status of a variable may be available than in the corresponding context of a top-down analysis. Hence, a bottom up analysis based upon variable states only cannot return the information to optimise the structure of the network. In the example of Fig. 8, if no information is available about the state of the arguments of P, a bottom up mode analysis of Select cannot return any information.

3.2. Ground dependency analysis

The loss of information due to the bottom up approach may be recovered by taking into account not only the states of variables, but also the relation about the states of distinct variables produced by the successful refutation of a predicate. This is possible through the notion of *ground dependency*, GD:

> a predicate P defines a ground dependency from arguments in $S = \{A_i, \ldots, A_k\}$ to an argument A_j, $A_j \notin S$, if any successful refutation of P where the arguments in S are bound to ground terms binds A_j to a ground term.

The notion of GD is fully independent of the variable states and the analysis to detect GDs does not require information about such states. Furthermore, the notion of GD allows the analysis to return significant information about the variable states because, provided that the groundness of a few variables may be determined, the state of other variables may be deduced through GDs.

From the declarative meaning of Select in Fig. 8, it is quite obvious that whenever the first argument is ground both the second and the third arguments, respectively the selected element and the resulting list, are ground. Furthermore, since Select may produce the first argument through a non-deterministic insertion of the second argument in the third one, the first argument will be ground, provided that both the second and the third ones are ground.

GDs are detected through the multiple occurrence of the variables in the arguments of a predicate, i.e. X and X_s in the first clause of Select and Y in the second one.

GDs define how predicates propagate ground information and they are strongly related to stream parallelism because this propagation has a critical role in the choice of the predicate to be used as the

producer. As matter of fact, the search space of the consumer is strongly reduced if the producer binds V to a ground term. More in general, the refutations of the predicates in the body of a clause should be ordered so that the i-th predicate defines a GD from the variables bound to ground terms by the previous $(i-1)$ predicates. Since, in this way, a partial order only may be defined, the compiler exploits proper heuristics to transform the partial order into a total one [32]. As an example, if several predicates bind the same variable to a ground term, then the producer is the one that binds the largest number of variables to ground terms.

A specialisation of the notion of GD is that of *deterministic ground dependency* DGD that extends the notion of functional computation in a logic program [16]:

> ground dependency is deterministic if for any set of ground bindings for the arguments in S, any successful refutation of P binds A_j to one ground term only.

The predicate Append3(X,Y,Z,K) in Fig. 9 is such that K is the result of appending the lists X, Y and Z and satisfies a given condition F_1.

The predicate Append defines three GDs:

1. from the first two arguments to the third one;
2. from the third argument to the first one;
3. from the third argument to the second one.

According to the declarative meaning of Append3, only the first GD is a deterministic one. This can be deduced because each step of a refutation exploits at most one clause of Append. Suppose now that A, B, C and D are ground when Append3 is invoked: then the three subgoals of Append3 cannot be concurrently executed because of the shared local variable E. If the producer of E is chosen on the basis of GDs only, either of the two Append literals may be chosen because they both produce a ground binding for E. If DGDs have been determined, then the first invocation of Append is chosen because it produces at most one ground binding for E. This choice largely reduces the search space and it may be considered as the optimal one, independently of the execution model.

If a predicate P defines a DGD, the instance of Dist in Net(P) sends any AF received from the input stream to at most one clause network.

```
Append([], X, X).
Append(cons(X, Y), Z, cons(X, V)) :- Append(Y, Z, V).
Append3(A, B, C, D) :- Append(A, B, E), Append(E, C, D), F1(E).
```

Fig. 9.

3.3. Type inference and share analysis

A bottom up polymorphic type inference system has been included in the compiler. The type inference system assigns types to functors and predicates through a program analysis that is independent of the functor names and of their arity. Type declarations are defined implicitly in the compiler and they are parametric with respect to the name of the constructors exploited in the type declarations.

Most type inference systems for logic languages are based upon those used in functional programming [30] and they can handle only regular structures that include objects with the same type. When a heterogeneous structure is analysed, either an error is signalled or α, the most general type that includes any type, is returned.

Our type system, instead, characterises heterogeneous data structures through the introduction of the type union constructor $\oplus$. As an example, if a list may include both Boolean values and characters, its type is list(Boolean $\oplus$ character). The type system assigns the type α to a structure if and only if it can include terms with any type.

The adoption of this type system is related to the definition of a bottom up analysis to detect the

```
Occ ([ ], ø).
Occ (cons(a,X), Z) :-Occ (X, T), Plus (T, s(ø), Z).
Occ (cons(b,X) Z):- Occ (X, Z).
Sumlist ([ ], ø).
Sumlist (cons(X ,Y) , K) :- Sumlist (Y, T ), Plus (X, T, K ).
Check (X, Y) :- Occ (X, L), Sumlist (Y, F), Larger (L, F).
```

Fig. 10.

sharing among variables due to a unification. This analysis detects literals of the same clause that cannot be refuted in parallel because they may share an unbound variable.

To improve the share analysis, its results have been integrated with those returned by the type inference system. In fact, two variables defined in terms of distinct elementary data types cannot share an unbound variable. As an example, no variable can be shared between a list of integers and one of strings.

To describe the proposed approach, consider the predicate Check(X,Y) in Fig. 10, that is verified if the number of occurrences of the character "a" in X is larger than the sum of elements of Y.

In the clause of Check, because of the shared variables L and F, one producer has to be chosen between Occ and Larger and one between Sumlist and Larger. Hence, the only opportunity is the parallel execution of Occ and Sumlist, but, even if a textual analysis does not detect any variable shared between them, a sharing may arise because of a previous unification.

In the case of Check, the types inferred for the arguments of Occ and Sumlist can guarantee that no sharing occurs at run-time. As a matter of fact, the type inference system detects that, whenever Occ succeeds, its first argument will be bound to a term of type list(character) while the type of the first argument of Sumlist is always list(natural). As a consequence, no sharing is possible between the two structures because they include elements with distinct elementary data types, i.e. no successful refutation can introduce a sharing. Hence, Occ and Sumlist can be refuted in parallel before Larger.

To show the importance of handling heterogeneous data structures, suppose that the clause

$$\mathrm{Occ}(\mathrm{cons}(\mathrm{true},X),Z) : -\mathrm{Occ}(X,Z)$$

is inserted into the program in Fig. 10. The type of the first argument of Occ becomes list(Boolean $\oplus$ character). Again, no sharing between Occ and Sumlist is possible. This can be deduced because the type system records the information about all the types of the elements in the two lists. A type system that is not heterogeneous does not enable the share analysis to deduce the absence of sharing between Occ and Sumlist because it assigns the type α, that may be instantiated to any data type, to the first argument of Occ.

3.4. K-ground dependency analysis

When stream parallelism is exploited, the producer generates a "complete" binding for a shared variable V before starting the consumer. This reduces the degree of parallelism when the consumer can perform some useful computation even when receiving a partial binding for the shared variable. Suppose that the shared variable is a list: in this case, the producer can send to the receiver a stream of elements of a list rather than a whole list. This cooperation is implemented through pipeline-AND parallelism and it requires a control mechanism similar to that of concurrent logic languages [33], where stream communication is fundamental. We point out that pipeline-AND parallelism is fully orthogonal to OR parallelism, because it is related to the definition of a single solution.

To determine if pipeline-AND parallelism can be exploited, a formal definition of "useful partial binding" is required, that describes the binding required by the consumer to perform some useful computation. The notion of k-ground dependency is a further refinement of a GD that approximates such a defini-

```
mm([],_,[]).
mm(cons(X, X_S),Y_S, cons(Z, Z_S)):- vm(X,Y_S, Z),mm(X_S,Y_S,Z_S).
vm(_, [], []).
vm(X_S, cons(Y,Y_S), cons(Z, Z_S)) :- ip(X_S,Y,Z), vm(X_S,Y_S,Z_S).ip([],[],0).
ip(cons(X, X_S), cons(Y, Y_S), Z) :- *(X, Y, Z_2), +(Z_2, Z_1, Z) ip(X_S,Y_S, Z_1).
```

Fig. 11. A matrix multiplication program

tion. A partially ground binding is useful if it allows the consumer to produce, in turn, ground bindings for other arguments.

A predicate P defines a *k-ground dependency*, k-GD, from the arguments in S to A_j if:

1. a ground dependency from S to A_j exists,
2. the arguments in S and A_j are structured terms,
3. for any substitution where the first k components of the arguments in S are ground, a successful refutation of P binds the first component of A_j to a ground term.

The analysis to detect k-GDs is applied if the type inference analysis returns a type list for at least two arguments of a predicate.

Consider, as an example, the program fragment in Fig. 11, where mm(X,Y,Z) is such that the matrix Z, a list of rows, is the product of matrices X, a list of rows and Y, a list of columns. In turns, both rows and columns are represented as lists. We assume that the predicates $+$, $*$ have the obvious meaning and that both define a GD from their two first arguments to the third one.

The type inference system deduces that the arguments of mm are two level structures, lists of lists. The predicate mm defines a GD from the rows and the columns of the input matrices to the rows of the resulting one.

The predicate vm defines GDs among second level elements of the structure, i.e. from the elements of the input matrices to those of the resulting one.

The analysis detects a 1-GD from the first two arguments of mm to the third one. This signals that a row of the resulting matrix can be bound to a ground term if the whole second matrix and the corresponding row of the first matrix are ground. vm defines each row of the third matrix as the product of a row of the first matrix for each column of the second one. The 1-GD from the first two arguments to the third one denotes that a ground binding for an element of the resulting row is produced provided that only the corresponding row and column in the input matrices are ground. Hence, a binding is useful for mm if the first subterm of the first argument of mm is ground and the second argument is ground. When supplied with such a binding, mm can bind to a ground term one element of its third argument.

To show how k-GDs make it possible to exploit pipeline-AND parallelism, consider the clause

$$\text{mmul}(X,Y,Z) :- \text{transp}(Y,T), \text{mm}(X,T,Z),$$

where transp(Y,T) is such that T is the transposition of matrix Y and a 1-GD from the first to the second argument of transp exists. In this case, if transp has been chosen as the producer of T, pipeline-AND parallelism can be exploited because of the 1-GDs from the first argument of transp to the shared variable T and of that from T to the other arguments of mm. These dependencies show that the partial ground bindings for T are useful for the consumer.

A further requirement to exploit pipeline-AND parallelism is that the partial bindings are consumed in the same order they are produced. This can be checked by the compiler because this order depends upon the one among clauses. In the example, the pipeline-AND optimisation can be applied if transp produces the columns of the transposed matrix in the obvious order.

4. Implementation of the model

A first implementation of the execution model and of the abstract interpreters has been developed. This implementation exploits a predicate-indepen-

dent version of Dist, Join, Split, Merge and RecHan. The size of the program of each process, that ranges from 5 to 10 Kbytes, could be reduced by optimising each process according to its position in the network.

4.1. Processor scheduling and flow

The experimental results have pointed out some critical problems related not only to the proposed model but to distributed memory systems in general.

Because of OR parallelism, a refutation of a goal may fire the refutation of a large number of subgoals. This results in the injection of a large number of messages into the interconnection network that leads to a sharp increase of the communication traffic. In turn, this can cause both congestion and hot spots. A hot spot [14,31] arises when a large percentage of the messages crosses the same link.

If we take into account that the latency of a communication is proportional to the number of links crossed by the corresponding message as well as to the number of conflicts on each link, i.e. of the messages that try to cross the same link simultaneously, it is obvious that congestion and hot spots strongly reduce the performance of the network and, hence, of the overall system.

Furthermore, both congestion and hot spots noticeably increase if several communications are non-local, i.e. between processes mapped onto PEs that are not directly connected or connected to the same routing device. In a direct interconnection network, non-local communications increase the computational load of the PEs that have to route the corresponding messages. In an indirect network, non-local communications increase the number of conflicts at each routing node. On the other hand, to map a complex process network onto the target architecture, non-local communications cannot be avoided [5,24].

To prevent a large performance degradation, we have adopted a control flow strategy, *congestion prevention by bounding* (CPBB). This strategy *bounds the number of messages flowing in the interconnection network in parallel* or, that is the same, *the number of processes that can send a message in parallel* [4]. The CPBB strategy seems more appropriate for asynchronous computational models than random routing [38] and it does not require proper hardware supports as virtual channels [13,14]. Furthermore, it makes it possible both to exploit locality in the application and to bound in advance the latency of a communication.

The CPBB strategy we have defined:

(a) *bounds the overall number of messages* flowing both in the overall interconnection network and in a subset of the network itself, i.e. it supports both global and local controls. Local control is fundamental to reduce hot spots;

(b) *chooses the processes that can send a message according to the status of the overall computation.* As an example, messages that can quickly lead to the production of a result have priority over those that still requires a large amount of computation before producing an output. Assigning a priority to each message is not a proper solution, because priorities depend upon the status of the receiver process as well.

It is worth noticing that, if most communications are not local, then several processes can be mapped onto the same PE so that the PE is not idle while one process is waiting because of a non-local communication. In this case, control flow should be integrated with the scheduling of the PE so that a process is scheduled for execution only if it is allowed to communicate. This avoids a too large context switching overhead. As discussed in the following, CPBB can achieve such an integration.

The CPBB strategy and the corresponding algorithm have been defined with reference to a class of process networks that includes those to support the implementation of logic programs. Here we describe the CPBB strategy at a high abstraction level and focus only on those aspects that are relevant to the

networks of interest, see Fig. 3. At first, structured networks with no recursion are considered.

CPBB is based upon the notion of a token that represents the permission to send a message, i.e. only a process holding a token can send a message. The number of tokens is fixed at program loading. If K is the number of tokens, at most K messages can be flowing in the network at any moment because only a process holding a token can send a message and because the process pairs the token with the transmitted message.

Each token is either *local* or *external* to a network. In the former case, the token supports the transmission of messages only within the corresponding network. In the latter, a token can cross different subnetworks and propagate AFs among the subnetworks. The tokens are managed so that the receiver of a message, i.e. of an AF, can start the refutation of the query coded by the AF as soon as it is received.

Starting from an initial allocation of token to processes, an algorithm distributes tokens among the processes so that the same results of an unconstrained execution are produced. The algorithm consists of *two steps*: *the forward and the backward distributions of tokens* that correspond to, respectively, the distribution of AFs among the networks and that of free tokens among the networks waiting to send/receive an AF.

The two steps alternate to guarantee that when a process sends an AF, the receiver is ready to execute the computation fired by that AF. The backward distribution of tokens in a network N terminates if and when all the outputs of N have been produced. To detect this event, each token T includes an information ES(T) about the execution status of the processes it has already crossed in the backward distribution. ES(T) makes it possible to determine if at least one computation is still suspended waiting for a token.

We notice that K, the number of tokens, depends upon the ratio between the degree of parallelism of the query network and the degree of parallelism that can be supported by the target architecture as well as upon the mapping of the network on the architecture. The value of K decreases as it increases the number of constrains the architecture imposes on the network. From another point of view, the larger the value of K, the lower the resource sharing due to the program mapping.

In the following, we assume that each process, or process network, N stores in a queue OutQ(N) the AFs it cannot send because no token is available. In(N) denotes the input stream of N that includes the AFs codifying the queries to be refuted by N.

4.1.1. Forward distribution

Because of the CPBB strategy, each message flowing in the network is either a pair ⟨AF,token⟩ or a token that is backward distributed to enable a process P to send messages in OutQ(P).

A process network N can receive a message ⟨AF,ET⟩ from In(N) only if it holds a free local token LT. Upon the reception of such a message, N holds the token ET that is, by definition, external for N. Hence, N can use ET to transmit one AF to the network ON whose input stream corresponds to the output one of N. LT enables N to start a computation and produce a result that will be either appended to OutQ(N) or transmitted to ON, i.e. appended to In(ON) if some external token is available. In both cases, LT becomes free because it is local to N. LT allows N either to receive a message from In(N) or to resume one of the computations fired by an AF previously received and that has been suspended because no token was available. These computations are detected through the token backward distribution.

4.1.2. Backward distribution

In the backward distribution phase, tokens are redistributed to enable the resumption of suspended computations. In the backward distribution the processes $N_1,\ldots,N_k$ belonging to a network N, trans-

mit tokens in the opposite direction with respect to AFs. Hence, tokens flow from the processes that transmit the AFs produced by N to those that receive the AFs from In(N).

As an example, in an OR network implemented as in Fig. 3, AFs flow from Dist to Join in the forward phase while tokens flow from Join to Dist in the backward phase.

During the backward distribution, any N_i crossed by a token LT either uses LT to transmit an AF in OutQ(N_i) or transmits the token backward if OutQ(N_i) is empty. The transmission of a message in OutQ(N_i) terminates the token backward distribution and it starts a new forward phase.

A backward phase terminates after distributing the tokens to any process or subnetwork of N that can produce an AF to be appended to OutQ(N).

As discussed in Section 2, because of OR parallelism, the execution model always requires *a termination detection algorithm independently of scheduling and congestion prevention*. This algorithm detects when the query network will not produce further AFs. The backward distribution is a generalisation of this algorithm that detects the subnetworks waiting to send an AF.

N can backward distribute any external token that it is currently holding when no computation is going on within N and no computation is suspended within N. In terms of AFs, these two conditions can be rephrased as follows: N can backward distribute an external token if no AF is flowing within N or waiting in a queue, i.e. OutQ(N_i) is empty for any process N_i belonging to N.

The backward distribution of external tokens allows other networks to produce queries to be received by N. Since N backward distributes an external token only after the termination of its computations, it privileges *the resumption of suspended computations with respect to the production of further queries for N*. In other words, N can receive an AF from In(N) only after computing all the solutions of the queries previously received.

The main advantages of this strategy are:

1. *low memory requirements* because it reduces the number of suspended computations;
2. *load balancing*: By favouring suspended computations, we reduce the time to produce the solutions for a given query and, hence, to start a computation in the networks that receive such solutions.

4.1.3. Recursion

To apply the CPBB strategy to networks that include RecHan processes, the distinction between local and external tokens has to be reconsidered because the recursive invocations produced by RecHan are solved by the same network, Net(P), including the RecHan process. Hence, RecHan needs an external token to append a message to In(P), but external tokens are handled by Dist and Join only because they are the processes of Net(P) that interact with the environment of Net(P).

This can be solved through another kind of token, an *indirect token*. An indirect token IT is local to the network Net(P) including the RecHan and, in the forward distribution, any process in Net(P) can consider IT as either an external token, as in the general case, or as a local token dedicated to the RecHan process. This means that a process of Net(P) can use IT to send a message to a RecHan only. Otherwise, the process handles IT as an external token.

The backward distribution phase handles indirect tokens as external ones.

4.2. Experimental results

The first experiments have been focused on a database query, the classical N-queens problem, that present several problems of more realistic applications, queries about array and matrix manipulation and the search of the shortest path between two nodes of a graph. Any kind of parallelism is present

(a)

```
same(C, D, S, P) :- dens(C, D), dens(S, P), >(D, P), >(20*D, 21*P).
dens( S, D):- pop(S, P), surf(S, A), Ratio (D, P, A).
pop ( china, 1000).
...
surf (china, 960).
...
```

(b)

PEs	kind of parall.	time (msec)
1	--	1043
6	RAP	253
14	stream	762/840
16	stream+RAP	164/192
17	stream+RAP+ parallel Merge	92/106
19	stream+RAP+OR2+parallel Merge	87/105
26	stream+OR3	195/210
28	stream+OR3+RAP	181/190
31	stream+RAP+OR3+parallel Merge	100/106

Fig. 12. a. Database program. The query is Same(X,Y,Z,K). b. Execution times.

in these programs. Several experiments have been developed by adopting the CPBB policy.

Because of the static approach, the number of PEs to be used in the experiments depends upon the process network returned by the compiler and it cannot be freely chosen as in a dynamic approach.

(a)

PEs/queens	kind of paral	time (msec)
20/4	stream+OR	243
23/4	stream+RAP +OR2	206
20/5	stream+OR	1026
23/5	stream+RAP+OR2	1006
22/6	stream+OR	4531
26/6	stream+RAP+OR2	4245

(b)

PEs/queens	kind of paral	time (msec)
20/4	stream+OR2	238
20/5	stream+OR2	728
20/ 6	stream+OR2	3630

(c)

PEs/queens	kind of paral	time (msec)
20/4	stream+OR2	315
20/5	stream+OR2	873
20/6	stream+OR2	4080

Fig. 14. N-queens program. a. Execution times b. Execution times with CPPB. c. Execution times with CPPB and random mapping.

Some the experiments have used a distributed memory system with 40 PEs, each including an T414 and a 256 Kbytes memory. The PEs are connected in

(a)

```
same (C, D, S, P) :-dens (C, D), dens(S, P), >( D, P), <(20*D, 21*P), avpr(C,R), avpr(S, T),
        ≥(R, T).
dens( S, D) :- pop( S, P), surf ( S, A) , Ratio (D, P, A).
avepr(C, D):- pop(C, P), prod( C, Gp), Ratio(Gp, P, D).
pop ( china, 1000).
...
surf (china, 960).
....
prod( china, 700).
```

(b)

PEs	kind of parall.	time(msec)
25	stream	842/891
29	stream+RAP	202/220
32	stream+RAP+parallel Merge	110/120
40	stream+RAP+OR2	212/232

Fig. 13. Second database program. The query is Same(X,Y,Z,K). b. Execution times.

a 4×10 mesh with wrap-around connections. Other experiments have used a distributed memory system with 32 PEs, each including a T800 and 2 Mbytes of memory. Both systems are rather simple and most communication related functions are implemented by software rather than at the hardware/firmware level as in current architectures. As an example, in both systems, messages are routed through proper processes replicated on each PE. This implies that all the reported performance figures can be considered as worst case ones.

4.2.1. Database programs

Given a database recording the population and the extension of a set of countries, at first we consider a query about all the pairs of countries whose average densities differ of less than 5%.

Fig. 12a shows the database program in the case of 25 countries, Fig. 12b reports the kinds of parallelism, the number of PEs that have been exploited and the corresponding execution time. Each time is an average over at least 8 runs. ORi means that the process Dist has been decomposed so that the facts about either the extension or the surface are managed by i processes mapped onto distinct PEs, i.e. i unifications have place in parallel. The experiments have shown that this improves the performance only if $i \leq 3$. This is due to the low number of physical links of each PE because, if $i > 3$, some of the communications among the processes produced by the decomposition of Dist are non-local. Parallel Merge denotes that the process Merge to implement AND parallelism has been decomposed into two processes to avoid memory overflows.

(a)

```
transform(X, [], X).
transform(X, Y, Z):-mol(Y, Y'),add(X, Y', Z).
mol([],[]).
mol([X | Y],[X1 | Z]):-X1 IS (3*X),mol(Y,Z).
add([],Y,Y).
add([X1 | X],[Y1 | Y],[W1 | W]):-W1 IS (X1+Y1), add(X,Y,W).
```

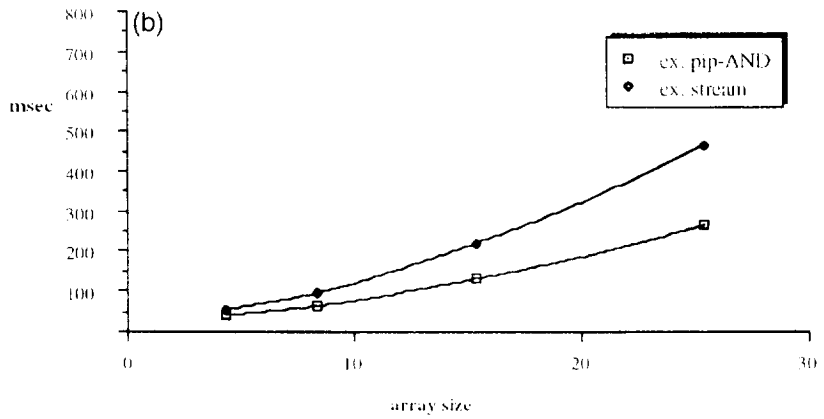

Fig. 15. a. Array program. b. Execution times.

A further benchmark has been considered, where the database includes the global national product of each country as well. In this case, the query is about all the pairs of countries whose average densities differ less than 5% and such that the average product per person of the first country is larger than that of the second one. The program and the execution times are shown, respectively, in Fig. 13a and Fig. 13b.

4.2.2. N-queens problem

In our experiments for this well-known problem, the largest value of N is 6. The execution times are reported in Fig. 14. Again, each time is an average over at least 8 runs. Also in these experiments, ORi means that the clauses with the same predicate symbol are managed by i processes. In the case of 6 queens, the application of the CPBB strategy reduces the execution time from 4245 msec to 3640 msec. Furthermore, the adoption of CPBB reduces the amount of memory to buffer messages as well as the size of the cactus stack managed by the RecHan processes.

4.2.3. Array and matrix product

Given two arrays of integers, the program shown in Fig. 15a multiplies each element of the first one for a constant and adds the resulting array to the second one. In the implementation of this program, OR, Restricted-AND and pipeline-AND parallelism may be exploited. We have compared the execution times that can be achieved through stream parallelism alone against those that can be achieved through both stream and pipeline-AND parallelism. As shown in Fig. 15b, the execution time halves when both kinds of parallelism are exploited with respect to the case where stream parallelism only is exploited.

The second program applies the same operation to matrices. The experimental results show that, with respect to the case where only pipeline-AND parallelism is exploited, no speed up is achieved by exploiting RAP in the producer network. The program in Fig. 16a has been executed on several matrices, by fixing the number of columns and by varying the number of rows from 3 to 7. In the case of matrices with 3, 4 and 5 columns, Fig. 16b and Fig. 16c show the execution times if pipeline-AND parallelism or stream parallelism alone is exploited.

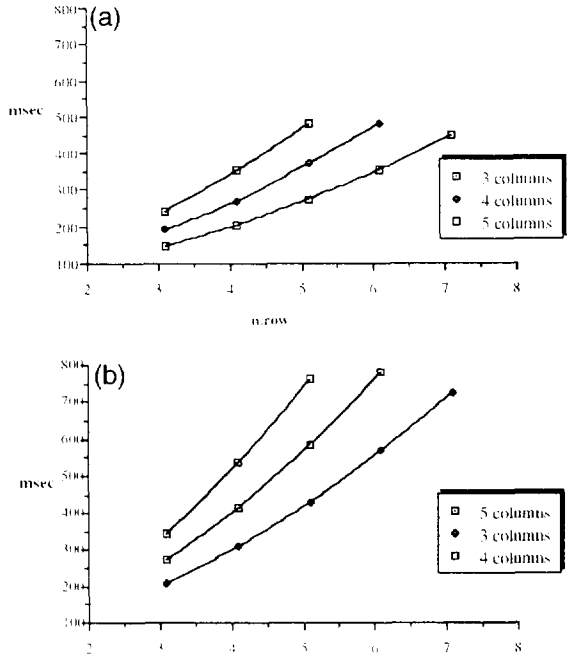

Fig. 16. a. Matrix program. b. Stream and pipeline-AND parallelism: Execution times. c. Stream parallelism: Execution times.

4.2.4. Shortest path problem

To evaluate the influence of the CPBB strategy on the final performance, a set of experiments has considered a program that computes the shortest path between any two nodes of a graph. Two versions of this program have been considered that are implemented, respectively, by 32 and 60 processes. Alternative mapping and routing strategies have been applied to the two versions.

The two routing strategies we have considered are an adaptive one, that chooses the path to route a

message according to the current load of the links, and an oblivious one [7], where the path to route a message between any pair of PEs is fixed.

Three mapping strategy have been applied:

(a) the *minimum distance mapping* (md). This strategy minimises the average distance between two communicating processes, i.e. the average number of links crossed by a message. To compute the average number of links, each communication is weighted according to the amount of transferred data and to its frequency;
(b) the *longest distance mapping* (ld). This mapping maximises the average distance between two communicating processes. The weight of each communication is determined as in (a);
(c) the *random distance mapping* (rd). A process is mapped onto a PE chosen at random according to a uniform distribution. This strategy does not attempt any optimisation but it requires no information about the amount of data exchanged in a communication or about the communication frequency.

To define the mappings (a) and (b), the amount of data exchanged among any pair of processes as well as the probability distribution of the interval of time between two successive communications have to be known. They have been obtained by program monitoring.

In all the experiments where the CPBB strategy is not adopted, the processes that are allowed to communicate or scheduled for execution are chosen by, respectively, the routing processes and the predefined scheduling algorithm of the Transputer.

The three parameters considered in the experiments are:

1. mul, the degree of multiprogramming of a PE;
2. the routing strategy: adaptive or oblivious;
3. the mapping strategy: ld or md or rd.

mul	mapping	routing	D
1	ld	adaptive	17.5%
1	ld	oblivious	38%
1	rd	adaptive	6%
1	rd	oblivious	21.5%
1	md	adaptive	17%
1	md	oblivious	18.5%
4	ld	adaptive	21.5%
4 (*)	ld	oblivious	-15%
4	rd	adaptive	15%
4	rd	oblivious	15%
4	md	adaptive	1%
4	md	oblivious	0%

Fig. 17. Execution times of the SPP, 32 processes version.

Concerning (1), in the case of the 32 process version, either 32 or 8 PEs are used and at most four processes are mapped onto each PE. The experimental results show that CPBB is an effective scheduling strategy for a PE. For each choice of mul, the mapping strategy and the routing algorithm, Fig. 17 shows D, the percentage improvement due the adoption of CPBB. If ex(CPBB) and ex(noCPBB) are, respectively, the execution times of the program if CPBB is applied and if it is not applied, then $D = (\text{ex(noCPBB)} - \text{ex(CPBB)})/\text{ex(noCPBB)}$.

Each time is an average over at least six executions. With the exception of (*), the CPBB strategy reduces, or at least it does not increase, the execution time, independently of the other parameters.

In case (*) in Fig. 17, the execution time increases because of the large number of non-local communications introduced by the ld mapping strategy. To support these communications, the PEs route messages for most of the time and they can devote a small percentage of time to the execution of processes and to the generation of AFs. Hence, no congestion arises and the CPBB strategy further increases the overhead.

Fig. 18 shows the results of the experiments for the 60 processes version. The execution times of this version are worse than those of the 32 processes version. By adopting the CPBB strategy, the execu-

mul	mapping	routing	D
2/3	ld	adaptive	85.9%
2/3	ld	oblivious	86.7%
2/3	rd	adaptive	76.3%
2/3	rd	oblivious	78.3%
2/3	md	adaptive	39.3%
2/3	md	oblivious	38.7%
8	ld	adaptive	71.2%
8	ld	oblivious	72.6%
8	rd	adaptive	69.0%
8	rd	oblivious	71.6%
8	md	adaptive	65.9%
8	md	oblivious	57.7%
4	ld	adaptive	82.1%
4	ld	oblivious	90.9%
4	rd	adaptive	65.7%
4	rd	oblivious	72.0%
4	md	adaptive	69.5%
4	md	oblivious	69.8%
16	ld	oblivious	65.9%

Fig. 18. Execution times of the SPP, 60 processes version.

tion times of this version are always lower than the corresponding ones in the 32 processes version. Besides those previously considered, we have evaluated a mapping where 4 PEs are used and at most 16 processes are mapped onto each PE. In this case, if the oblivious routing strategy is adopted, a message crosses, at most, two links. The reduction of the execution time in this case confirms the effectiveness of CPBB as a scheduling strategy.

The largest improvements in the version with 60 processes are due to a larger number of messages or, in other words, to a finer process grain. As a matter of fact, the 32 processes version has an almost optimal process grain for the considered architecture. It is worth noticing that, for any choice of the parameters, the execution time in the case (CPBB + ld mapping) is lower than that of the case (noCPBB + md mapping). Obviously, it is simpler to adopt the CPBB strategy rather than defining the md mapping.

Lack of space prevents a more extensive analysis of other experiments with a "hybrid" solution where the CPBB strategy has been applied only to those subnetworks where congestion may arise. Even in these cases, performance improvements similar to those previously described have been achieved.

5. Conclusion

To the best of our knowledge, ours is one of the first approaches to the execution of logic programs on distributed memory systems where a static execution model is integrated with static analyses to optimise the output process network.

At this stage of the experiments, preliminary conclusions only can be derived. The first results suggest the adoption of the proposed approach in the case of a database program or, however, in the case of "all solutions" applications. Even if side effects constructs such as *cut* and *not* have been implemented in the proposed model, the effectiveness of the solution in the case of "single solution" applications is more questionable. More powerful static analyses or further optimisations of the predefined processes have to be investigated for these problems.

The use of the CPPB algorithm has pointed out the importance of dynamic resource scheduling to reduce the performance losses due to sharp increases of the communication load. These sudden changes in the communication load cannot be foreseen at compile time and require a dynamic solution.

References

[1] S. Abramsky and C. Hankin (eds.), *Abstract Interpretation of Declarative Languages* (Ellis Horwood Ltd., 1987).

[2] J.K. Aggarwal and A.S. Lee, A mapping strategy for parallel processing, *IEEE Trans. on Computer* 4 (1987) 433–442.

[3] Arvind, K.P. Gostelow, The U-interpreter, *IEEE Computer* 2 (1982) 42–49.

[4] F. Baiardi, A. Candelieri and L. Ricci, Congestion prevention by bounding in distributed memory systems, in: A. De Gloria, M.R. Jane, D. Marini (eds), *Transputer Applications and Systems 94* (IOS Press, Amsterdam, 1994) 843–859.

[5] F. Berman and L. Snyder, On mapping parallel algorithms into parallel architectures, *Proc of 1984 Int. Conf. on Parallel Processing* (Bellaire, Michigan, 1984) 307–309.

[6] L. Bic, A data driven model for parallel interpretation of logic programs, *Proc. of Int. Conf. Fifth Generation Computer Systems* (1984) 517–523.

[7] A. Borodin and J.F. Hopcroft, Routing, merging and sorting

on parallel models of computation, *Journal of Computer and System Science* 1 (1985) 130–145.

[8] M. Bruynooghe G. Janssens, B. Demoen and A. Callebaut, Abstract interpretation: Towards the global optimization of prolog programs, *Proc of 4th IEEE Symp. on Logic Programming* (San Francisco, 1987) 192–204.

[9] J.H. Chang, High performance execution of prolog program based on a static data dependency analysis, Technical Report UCB/CSD 86/263, Univ. of California, Berkley, 1986.

[10] J.S. Conery and D.F. Kibler, AND parallelism and non-determinism in logic programs, *New Generation Computing* 3 (1985) 43–70.

[11] J.S. Conery, Binding environment for parallel logic programs in non-shared memory multiprocessors, *Proc of 1987 Int. Conf. on Parallel Processing* (1987) 457–467.

[12] P. Cousot and R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, *Proc of 4th ACM Symp. on Princ. of Progr. Languages* (1977) 238–252.

[13] W.J. Dally and D.S. Wills, Universal mechanisms for concurrency, Proc. of PARLE'89; *Lecture Notes in Computer Science Vol. 365* (Springer, Berlin, 1989) 19–33.

[14] W. Dally, Virtual-channel flow control, *Proc. of 17 Int. Symp. on Computer Arch.* (1990) 60–68.

[15] W. Dally, Network and processor architecture for message driven computers, in: R. Suaya and G. Birtwistle (eds.), *VLSI and Parallel Computers* (Morgan Kaufmann, New York, 1990) 140–219.

[16] S.K. Debray and D.S. Warren, Functional computations in logic programs, *ACM Trans. on Prog. Lang. and Systems* 3 (1989) 451–481.

[17] D. DeGroot, A technique for compiling execution graph expressions for restricted AND-parallelism in logic programs, *Journal of Parallel and Dist. Computing* 5 (1988) 494–516.

[18] B.S. Fagin and A.M. Despain, The performance of parallel prolog programs, *IEEE Trans. on Computer* 12 (1990) 1434–1445.

[19] R. Giacobazzi and L. Ricci, Pipeline optimization in AND parallelism by abstract interpretation, *Proc of 7th Int. Conf. on Logic Programming* (MIT Press, Boston 1990) 291–305.

[20] R. Giacobazzi and L. Ricci, Detecting determinate computations by a bottom-up abstract interpretation, *Proc. of European Symposium on Programming* (1992) 167–181

[21] R. Hasegawa and M. Amamiya, Parallel execution of logic programs based on dataflow concepts, *Proc. of Int. Conf. on Fifth Generation Computer Systems* (1984) 507–516.

[22] M.V. Hermenegildo, An abstract machine for restricted AND-parallel execution of logic programs, *Proc. of 3rd Int. Conf on Logic Programming* (1986) 25–39.

[23] M.V. Hermenegildo, R.I. Nasr, Efficient management of backtracking in AND-parallelism, *ACM Trans. on Programming Languages and Systems* 3 (1989), 40–53.

[24] A.J.G. Hey, Experiments in MIMD parallelism, *Proc. of PARLE'89; Lecture Notes in Computer Science Vol. 365* (Springer, Berlin, 1989) 28–42.

[25] Y. Lin and V. Kumar, An execution model for exploiting AND-parallelism in logic programs, *New Generation Computing* 5 (1988) 393–425.

[26] Y. Lin and V. Kumar, A parallel execution scheme for exploiting AND-parallelism of logic programs, *Proc of 1986 Int. Conf. on Parallel Processing* (1986) 972–975.

[27] H. Mannila and E. Ukkonen, Flow analysis of prolog programs, *Proc of 4th IEEE Symp. on Logic Programming* (1987) 205–214.

[28] K. Marriott and H. Sondegaard, Bottom-up abstract interpretation of logic programs, *Proc of 5th Int. Conf. and Symp. on Logic Programming* (1988) 733–748.

[29] D.L. Mc Burney and M.R. Sleep, Transputer based experiments with the ZAPP architecture, *Proc. of PARLE '87; Lecture Notes in Computer Science Vol. 258* (Springer, Berlin 1987) 242–259.

[30] R. Milner, A theory of type polymorphism in programming, *Journal of Computer and System Science* 3 (1978) 348–375.

[31] G.F. Pfister and V. Alan Norton, Hot spot contention and combining in multistage interconnection networks, *IEEE Trans. on Computer* 10 (1985) 943–948.

[32] L. Ricci, Compilation of logic programs for massively parallel systems, Ph.D. Thesis, Università di Pisa, 1990.

[33] E. Shapiro (ed.), *Concurrent Prolog Collected Papers* (MIT Press, Boston, 1987).

[34] B. Schwinn, G. Barth and C. Welsch, RAPiD: A data flow model for implementing parallelism and intelligent backtracking in logic programs, *Proc. of PARLE'89; Lecture Notes in Computer Science Vol. 365* (Springer, Berlin 1989) 115–132.

[35] D.J. Troy, C.T. Liu and W. Zhang, Linearization of nonlinear recursive rules, *IEEE Trans. on Software Engineering* 9 (1989) 1109–1119.

[36] E. Tick, A performance comparison of AND and OR parallel logic programming architecture, *Proc. of 5th Int. Logic Programming Conf.* (1989).

[37] P. Tinker and G. Lindstrom, A performance oriented design for OR-parallel logic programming, *Proc. of 4th Int.Logic Programming Conf.* (1987) 601–615.

[38] L.G. Valiant, Optimality of a two phase strategy for routing in interconnection networks, *IEEE Trans. on Computer* 9 (1983) 861–863.

[39] D.H.D. Warren, OR-parallel execution models of prolog, *Proc. of Theory and Practice of Software Development; Lecture Notes in Computer Science Vol. 250* (Springer, Berlin 1987) 243–259.

Fabrizio Baiardi graduated in Computer Science at the University of Pisa in 1980. He has been an associate researcher with the Department of Informatics, University of Pisa, since 1984. Currently he is an associate professor with the same department. His research interests include programming environments and operating systems for MIMD massively parallel systems. He has been the coordinator of the research activities on highly parallel operating systems of the national project on Parallel Architecture of the National Research Council (CNR).