A DISTRIBUTED IMPLEMENTATION FOR PARALLEL LOGIC PROGRAMMING

Antonio Brogi (*), Anna Ciampolino (**), Evelina Lamma (**), Paola Mello (**)

(*) Dipartimento di Informatica
Università di Pisa
Corso Italia, 40
56125 Pisa - Italy

(**) Dipartimento di Elettronica, Informatica e Sistemistica - DEIS
Università di Bologna
Viale Risorgimento, 2
40136 Bologna - Italy

Abstract

The distributed implementation of a new communication model for parallel logic programming is presented. The main novelty with respect to STREAM-parallel logic languages is that AND parallel processes do not share variables, and inter-process communication is performed via multiple-headed clauses. A compilation technique on an extended Warren Abstract Machine is described where new instructions and data structures are introduced for process creation and communication, and control of non-determinism. To show that this model is suitable for distributed architectures, a first prototype developed on a transputer-based architecture is presented.

Introduction

In recent years, many efforts have been devoted to the definition of new parallel logic programming languages. In the process interpretation of logic programming [1] a goal is viewed as a network of processes communicating via shared variables, while the clauses of the program correspond to alternative definitions of the processes. Although shared variables among parallel logic processes provide a basic communication mechanism, they are not adequate for synchronizing concurrent computations.

In STREAM-parallel languages [2], constraints on the unification mechanism have been introduced to synchronize processes via shared variables. The sharing of variables between parallel processes reprences, however, a centralization point in the resulting computational model. In the context of the Japanese project for Fifth Generation Computer Systems, where several implementations of concurrent logic languages have been developed, the sharing of variables between parallel processes is that performance evaluations do not seem satisfactory, even when sophisticated compilative techniques are exploited [4].

Alternative approaches with respect to the sharing of variables propose clauses with a conjunction of atoms in the head (multiple-headed clauses) [5] or events [6] for synchronizing parallel processes in logic programming. Multiple-headed clauses are defined to be simultaneously applied to matching conjunctions of parallel goals.

In this paper, we address the problem of the distribute implementation of a new communication model for parallel logic programming, where AND processes do not share variables, and inter-process communication is performed via multiple-headed clauses. The distributed nature of the resulting language (named ROSE [7]), has been formally demonstrated by the definition of a true concurrent operational semantics.

This paper mainly focuses on implementation issues by presenting the process model of ROSE and its implementation on a distributed architecture. Avoiding the sharing of variables between AND processes notably simplifies the realization of a distributed implementation with respect to STREAM-parallel languages. However, the unification phase is more complex and, correspondingly, a different structure for the process network with respect to STREAM-parallel languages is adopted.

In particular, the AND/OR tree of processes modeling parallel logic programming [8] is here a graph, while a tree of "unification" processes is introduced to support incremental multiple-head unification.

The implementation of ROSE is obtained in terms of compilation on an abstract machine which is a rather natural extension of the one for Prolog (Warren Abstract Machine - WAM [9]). Essentially, in the extended WAM new instructions are introduced for process creation and communication, and control of non-determinism. Accordingly, new data structures are added for process handling.

The architectural scheme is a parallel Multiple Instruction Multiple Data (MIMD) machine with distributed memory, and consequently the parallel model is based on message-passing. In particular, for the first prototype, we have adopted an architecture based on the transputer technology [16].

Parallel Logic Programming

A logic program is a collection of definite clauses. A definite clause is a logical implication of the form:

\[ A \leftarrow B_1, \ldots, B_n \]

where the \( B_i \) for \( i=1, \ldots, n \) are atomic formulae and \( n \geq 0 \). The consequent of a clause \( (A) \) is called head, and the antecedent \( (B_1, \ldots, B_n) \) body. Atomic formulae have the following form:

\[ p(t_1, \ldots, t_m) \]

where the \( t_m \) are terms. Variables are universally quantified and denoted by uppercase letter.

Definite clauses can be read declaratively as logic assertions or procedurally: i.e. "to solve a goal matching \( A \), solve the subgoals \( B_1, \ldots, B_n \)."

Since the goal can match with different clauses, logic programming is non-deterministic. Even if Prolog [11] is sequential, logic programming is suitable for parallel evaluation. In fact, the evaluation of subgoals in a conjunction can be performed concurrently (AND parallelism) and all the clauses matching a subgoal can be applied in parallel (OR parallelism). With respect to AND parallelism, we can distinguish two different forms:

1. Restricted or independent AND parallelism. We evaluate in parallel some subgoals in conjunction which do not share variables;
2. STREAM AND parallelism. We evaluate concurrently calls which can share variables, with the value of the shared variables communicated incrementally between the calls.

Since AND parallelism is very difficult to implement in the presence of non-determinism, some means for constraining the non-determinism is necessary.

ROSE

In this section an overview of ROSE is given. A complete description of the language, including its semantics and several programming examples, can be found in [7].

ROSE is a proper extension of Horn Clause Logic, where multiple-headed clauses are allowed:

\[ \leftarrow A_1, \ldots, A_n \leftrightarrow B \]

The \( \leftarrow \) parallel composition operator may occur both in consequent and the antecedent of a clause. The language forbids the sharing of variables among parallel goals in the body of a clause to always obtain independent AND processes.

In order to apply a multiple-headed clause, like \( \leftarrow \), a parallel composition of atoms, \( A_1, \ldots, A_n \) say, unifying with the clause heads has to occur in the
current goal, so that there exists a substitution (most general unifier) 
\( sgn((A_1 + ... + A_n): (A_1': + ... + A_n')) \).

For example, given the clause: \( a(X) + b(Y) < g(1) + g(2) \) the goal:
\( <-a(1) + a(2) + b(3) \) can be reduced to \( <-g(1) + g(2) + a(2) \) by unifying the head
(\( a(X) + b(Y) \)) with (\( a(1) + b(3) \)), or to \( <-g(2) + b(2) + a(1) \) by unifying the
same head with (\( a(2) + b(3) \)).

The focus of the present paper is on the distributed aspects of the language,
thus only the parallel composition operator (**") will be considered,
omitting the description of the sequential part of ROSE.

The definition of ROSE has been extended in [12], where a committed-
choice behaviour is introduced to control non-determinism. Don’t care non-
deterministic computations are defined by means of a guard mechanism as in the
family of STREAM-parallel languages [2].

Guarded multiple-headed clauses have the form:

\( (<**>) \) Head <- Guard \( I \) Body

where Head and Body are parallel conjunctions of atoms (i.e. \( A_1 + ... + A_n \))
and Guard is composed of system predicates only.

The meaning of the guard is that a clause like \( (***) \) is applicable (or candidate) to a
given goal if and only if both the head unification succeeds and the guard is satisfied.
For instance, the clause:
\( a(X) + b(Y) < X \neq Y \ I \) Body

can apply to the goal \( <-a(3) + b(1) \), but not to the goal \( <-a(1) + b(1) \).

A simple programming example is the well-known problem of the dining
philosophers.

Each philosopher is defined by the following two clauses:

\( \text{philo}(X) + \text{fork}(Y) + \text{fork}(Z) < -Y = X, Z = X + 1 \ I \ \text{eats}(X) \)
\( \text{eats}(X) < -X = X, Z = X + 1 \ I \ \text{philo}(Y) + \text{fork}(Y) + \text{fork}(Z) \)

For a symposium with \( n \) philosophers, the initial goal is:
\( <-\text{philo}(1) + ... + \text{philo}(n) + \text{fork}(1) + ... + \text{fork}(n) \)

Notice that in the second clause of the example the variable \( Y \) is not bound
to \( X \) (and therefore \( X \) is not shared between philo and fork goals) since at runtime
\( X \) will be bound to a constant.

A Fully Distributed Execution Model

In this section we define a fully distributed execution model for ROSE. The
model is based on AND and OR processes which execute asynchronously
and communicate by message passing.

Since the language forbids the sharing of variables among parallel goals
in the body of a clause, AND processes are always independent. The
innovative issue is the implementation of the multiple-head unification
which is the basic mechanism of ROSE for process interaction. In particular,
an AND/OR graph of processes and an incremental, parallel unification
scheme are introduced.

AND/OR graphs of processes

In order to describe the process model of ROSE, the AND/OR process tree
of parallel logic programming [8] is replaced by a directed acyclic AND/OR
tree.

The initial goal (<-c(1) + ... + g(k)) is represented by an AND node with k OR nodes
as children, one for each atom composing the goal.

Each of these OR nodes is thus labeled by some atom \( g_i \) and has a child AND
node for each clause such that one atom of its head unifies with \( g_i \).

As in all “committed-choice” languages [2], in ROSE OR-parallelism is
limited to head unification and guard evaluation, which is sequential for the
sake of simplicity.

"Committing" to at most one OR process simplifies the AND/OR graph
structure and therefore the implementation. In practice, only the committing
OR process will continue the computation by generating the corresponding
AND processes, while sibling OR processes can be killed.

However, the commit implementation is more complex with respect to

\[\text{Figure 1: An AND/OR graph}\]

STREAM-parallel languages due to multiple-head unification. Whenever an
OR process tries to commit, after the successful execution of the clause
guard (candidate OR process), it has to send an explicit notification to more
than one AND process, i.e. to all its parent nodes.

For example, with reference to figure 1, both the candidate OR processes
labeled respectively by \( a(1) + b(3) \) and \( a(2) + b(3) \) succeed in
the guard execution, try to notify the commitment to \( b(3) \), but of course only one of them can successfully commit
by reducing the goal: \( <-a(1) + a(2) + b(3) \).

A two-phase lock protocol between the candidate OR processes and the
corresponding AND processes is provided in order to guarantee that only one
AND process can successfully commit.

Moreover, to avoid deadlock, an order in sending messages during the lock
protocol is imposed.

Incremental unification

An incremental parallel unification scheme is introduced for creating the
AND nodes of the AND/OR graph, and their associated OR processes.

The process model is in theory, a dynamic creation of different,
communicating processes, corresponding to the same clause \( C \), each one
responsible for a single head unification. For each clause, a “manager”
process is created to perform multiple-head unification.

Each AND process, responsible for an atomic goal \( g \), sends a message (g)
to each manager process of a clause \( C = h_1 + ... + h_n \ < \text{Goal Body} \).

After receiving the message, the manager of \( C \) creates a new child process
in order to perform the unification of \( g \) with \( h \) and routes the received message
to all the already created children. They, in turn, can create new children
for subsequent unifications at new message receptions. Such incremental
unification corresponds to a tree of “unification” processes, each one
responsible for a single head unification. Let us consider, as an example,
the clause \( C: a(X) + b(Y) < \text{true} \ I \ g(X) + g(Y) \).

Initially, a manager process, \( N_0 \), is created for \( C \). \( N_0 \) waits for some atom which may unify with \( a(X) \) or \( b(Y) \).

Given the parallel goal: \( <-a(1) + a(2) + b(3) \), three AND processes, \( P_1 \), \( P_2 \), and
\( P_3 \), are created for \( a(1) \), \( a(2) \) and \( b(3) \) respectively.

If \( P_1 \) sends the corresponding message (\( a(1) \)) to \( N_0 \), \( N_0 \) creates one child,
\( N_1 \), for unifying \( a(1) \) with the \( a(X) \) variable. If \( P_2 \) sends the corresponding message (\( a(2) \)) to \( N_0 \), \( N_0 \) creates another child process \( N_2 \) for unifying \( a(2) \) with the first head of \( C \), and routes the message also to the
already created children \( N_1 \) and \( N_2 \). \( N_1 \) and \( N_2 \), in turn, create the child processes \( N_4 \) and \( N_5 \) to perform
the new head unification. Thus, the unification tree grows as shown in figure
2, where: \( [ ] \) denotes the alternative command, and \( \text{"?A"} \) indicates that the
process waits for an atom \( A \).

Notice that processes \( N_4 \) and \( N_5 \) both complete the multiple-head unification,
and therefore they correspond to the OR processes of the AND/OR graph.

They start the guard evaluation and only one of them will commit. All the other
nodes of the tree represent processes which have not yet completed the
multiple-head unification and wait for other messages.

Let us note that the configuration of the tree can depend on the scheduling
of the messages. In the example above, we assume the order a(1), a(2), b(3) in receiving the messages. A different order would generate a different unification tree. However, the following property holds: given a set of messages (goals) and a clause C, whatever the order of messages, the leaves of the tree of depth equal to the number of heads of C, i.e. 2 in the case of the example above, which correspond to OR processes, are always the same.

The Rose Abstract Machine

Figure 2: A unification tree

The Warren Abstract Machine (WAM) [9] is a multiple-stack abstract machine designed for the efficient execution of Prolog. Prolog programs can be efficiently implemented by compilation into WAM instructions. The WAM model defines three stacks for storing binding for variables, environments, and choice-points and consistently handling backtracking. An environment is associated with each clause and acts as a recording for the clause. An environment holds variables (called permanent) and bookkeeping information. A choice-point holds arguments passed to a non-deterministic procedure and backtracking information.

In the following we present the abstract machine developed for the parallel execution of ROSE. The ROSE abstract machine draws ideas from the WAM but there are some significant differences in the code generated due to the process model and the need for a multiple-head unification.

When implementing parallel and distributed logic programming models, the environment representation has to be deeply revised (see [13,14]). In our particular case, the notion of environment of a clause as defined for standard WAM has to be further revised, since we deal with extended clauses, where we can have multiple heads and no variable sharing among AND parallel goals. In particular, environment variables are only those shared among different heads of a clause or those shared between one (or more) head(s) and goal(s) in the body.

Since in a parallel implementation of logic programming permanent variables represent data shared among processes, the revised notion of environment variable in ROSE implies that each AND process cannot share any environment variable with any other AND process, but only with some OR process. Moreover, environment variables can be shared among the processes belonging to the same unification tree.

Notice that the commit-choice nature of the language does not require the use of multiple environments to follow alternative paths originated from alternative clauses, which would assign different values to the parent variables.

In a fully distributed implementation this sharing of variables implies that both forward and backward unification have to be performed properly through dereferencing and message passing [15]. Forward unification takes place whenever a unification process is created and successfully performs unification. Input values for unification are contained in the message sent by the corresponding AND process and routed, if needed, by other unification processes. Backward unification takes place whenever an OR process successfully ends its body. Output values for variables are therefore returned through message passing to the involved AND processes.

The distributed implementation of backward and forward unification requires a great number of communications and therefore a high overhead. However, thanks to the ROSE computational model that forbids the sharing of variables among parallel goals, backward unification can be performed only for the variables of the top-level goals and variables bound to them. We will refer to those variables as "output" variables. In the most general case, output variables can be determined only at run-time, since they depend on the top-goal. For this reason, we extend the representation of environment variables by an additional tag, in order to distinguish output variables. This optimization, not applicable in "STREAM"-based languages, implies a significant improvement of the distributed implementation of ROSE, since backward unifications are performed only for tagged variables and therefore the communication load is highly reduced.

Of course, data structures of the WAM have been extended to deal with AND, manager, unification and OR processes by introducing suitable descriptors which are described in the next section.

New instructions are introduced with respect to standard WAM to deal with processes.

Each goal connected by the parallel composition operator (+) is compiled into an AND process creation by using the new create_and instruction. The new create_or/create_unif instructions are introduced to create respectively OR and unification processes. All the create instructions specify as arguments the label of the code to be executed.

Each AND process will execute a send-msg pred_N instruction to send the corresponding goal to each manager process of interest, and then suspends. If hl,...,hn <- GIB, is the clause associated with a manager process P, P executes a wait [hl,L1,...,hn,Ln] instruction waiting, in a non-deterministic way, for some message of interest. If the message is selected, a jump to the label Lj is performed.

In the compiled code, each guard is followed by the new commit instruction.

The Real Implementation

The architecture

The chosen architectural scheme for the real implementation of ROSE is a parallel MIMD machine with distributed memory, and the process interaction scheme is based on message-passing. In particular, we will refer to a Meiko Computing Surface architecture, which is based on the transputer technology [16]. The configuration adopted for the first implementation of ROSE is composed by sixteen transputers, each one with 4Mbytes of local memory. In addition to the point-to-point interconnection network between transputers, the Meiko Architecture presents a monitoring bus (Supervisor Bus) which connects all transputers together. It represents an alternative way to reach each transputer, instead of transputer links. It can be used, for example, to observe the dynamic evolution of computational load on processors, without perturbing the communication traffic in the system.

With reference to the programming tools and languages to be used, the OCCAM language [17] well fits into the transputer architecture, but it is too limited since it allows neither the dynamic creation of processes nor asynchronous communication. In the first ROSE implementation we have rather used CSTools [18], a more flexible programming tool obtained by extending the C language with primitives for process creation and interaction. The CSTools library offers a set of primitives that reduce constraints on the communication between processes, and overcome the OCCAM channel concept by associating asymmetric, bi-directional (synchronous or asynchronous) ports with processes. Therefore, the translation of the communication protocols between processes of our distributed implementation scheme into CSTools primitives has been rather natural.

The implementation of the process model

Initially, a top-level AND parallel goal is spawned in a number of AND processes and one manager process is created for each multiple-headed clause. After the creation, each AND process sends a message containing its corresponding goal to each manager process of interest and then suspends, waiting for a success or a failure message. In the case of success, the resulting message will contain the bindings for the "output" variables. Each unification process, manager included, at message reception, creates a new process to perform the next head unification. When the last head unification is successfully performed (by an OR process), the guard and possibly the commit protocol is executed.

As in [19], our execution model is characterized by an "eager" creation of processes, which is in contrast, for example, with the policy adopted in [20] and [21] where the number of parallel activities is naturally limited to the
number of physical processors. Clearly, our choice implies, in the worst case, a combinatorial explosion of processes but the programmer can directly control the parallelism, since ROSE supports both sequential and parallel computations.

For the first prototype, we rely on the scheduler of each transputer for scheduling processes allocated on the same node. In the future, we plan to investigate more sophisticated policies by explicitly implementing scheduling algorithms.

Each process is represented by a descriptor, which stores both static and dynamic information. We associate a single, independent process (the Manager) to each clause of a ROSE program. When receiving a message from an AND process, the Manager generates one or more unification processes with the same descriptor structure. In particular, each manager or unification process descriptor contains the following information:

1) the program instruction pointer (P);
2) a list of identifiers (FatherList) of the AND processes whose messages have been successfully unified. A copy of this structure is passed to each new process created by the unification/manager process (suitably upgraded with the new unification information);
3) a list (ChildList) of the processes corresponding to the child nodes in the unification tree, in order to perform the routing of messages, together with their unification state in order to perform a "more intelligent" routing of the messages. In fact, the incremental unification mechanism implies a very heavy message traffic across the network. An efficient routing of messages between processes in the tree can be obtained by forwarding messages only to those processes which can really consume them, i.e. that can unify.

In the case of OR processes, the creation mechanism is mainly the same, but descriptors have a different structure. In particular, each OR process descriptor contains the following information:

1) a list of identifiers (FatherList) of the AND processes whose messages have been successfully unified, in order to propagate the two-phase lock protocol (in case of COMMIT) and to propagate success (and "output" variable bindings) or failure results;
2) the program instruction pointer (P);
3) a counter (count), as in [10], representing the number of AND processes that have this node as their father, for determining success. Each time one success message is received, the counter is decremented. If the counter has reached zero, the success is propagated from the OR process to the father AND processes.

Let us notice that for success and failure propagation after commitment, OR processes need to know only the AND processes whose messages have been successfully unified, thus forgetting any information about their (unification process) creators.

Finally, the AND process descriptor contains:

1) the identifier of the father process (Father), in order to propagate success or failure;
2) the program instruction pointer (P);
3) a flag (Commit) that is set when one of the corresponding OR processes commits and a flag (Lock) to perform consistently the two-phase lock protocol;
4) a list of manager process identifiers (ManagerList) of managers whose clause heads can unify with the AND goal predicate. This kind of information can be statically obtained.

One of the key issues is a correct distributed implementation of the commit phase that must determine the exclusive assignment of a pool of AND "resources" to only one OR "consumer" process. We have designed a protocol that guarantees a deadlock-free commitment by maintaining a strict global ordering between AND processes derived from their identifiers.

More in detail, an OR process which has successfully evaluated its guard, sequentially sends:

1) a "reservation" to all its father AND processes following their global order;
2) a "confirmation"/"cancellation" to each reserved father AND process if step 1 succeeds/fails.

During the commit phase, an AND process waits for messages sent by its child OR processes. When a "reservation" message is received, the AND process suspends until a "confirmation"/"cancellation" from the same reserving process arrives. This implies that any AND process sending a reservation message to it will be blocked. When a "confirmation" message arrives, the other potential OR consumers of the reserved AND resource are killed. If a "cancellation" message arrives, the AND resource is made free and new requests are taken into account. The pseudo-code representing the implementative scheme of this mechanism is reported in Figure 3.

When an OR process P commits, all its ancestors in the unification tree, except the manager, become unuseful since they will create only processes that consume at least one of the resources definitely assigned to P. For this reason, we choose to introduce a sort of "garbage collection" mechanism on the unification tree based on the forward propagation of kill messages from the manager towards the tip nodes.

**Allocation**

Due to the limited number of processors and links per node available for the communication (four), proper allocation strategies are investigated to efficiently map the processes on a transputer network. In order to determine a good process allocation, the following factors have been taken into account:

- The granularity of processes;
- The coupling degree, which reflects the frequency of interactions between couples of processes;
- Architectural features, such as the number of available processing elements, and the interconnection network characteristics.

In the first prototype of ROSE, built for testing the model rather than deeply discussing performance evaluations, we have chosen to map all AND processes generated by an OR process P on the same physical node of P. Two reasons motivate this choice. First, the computational load of an OR process after the creation of its AND children is very light since it only waits for a result deriving from its successors. The second reason is due to the sharing of environment variables between the OR process and its children. If they are allocated on the same node, this is easily implementable without message passing.

With reference to manager processes, our decision is to allocate them on separate nodes, whenever possible, together with all the unification processes of the tree. There is, in fact, a high coupling degree between processes in the same unification tree, while the computational load of each unification process is light. OR processes, instead, are allocated on separate nodes since they carry on the greater part of the computation that consists of the guard evaluation and the body code execution.

However, if availability of physical nodes is not a problem, the allocation of manager and unification processes can take place on different nodes, in order to completely exploit the high parallelism degree of the incremental unification mechanisms. We will evaluate response times of different strategies on the basis of some application examples.

From the implementative point of view, the creation of a new process on a remote node implies the presence of an active process on the remote node whose function is to create processes locally. This is due to the lack of programming tools for the dynamic remote creation of processes. Therefore, on each node which can be a potential site of process execution, such a process (Node Manager) must exist.

With reference to the allocation policy, new OR process allocation is determined by a static circular policy, not the best but the less expensive in terms of computational load. The alternative, in fact, is to support allocation decisions with dynamic load monitoring, in order to choose the allocation site depending on the current load conditions. In this case, system monitoring processes have to be introduced. Each ROSE process has to inform the dedicated monitoring process of its activation/termination in order to make the monitoring processes able to decide about allocation strategies. This monitoring mechanism can be completely centralized, or distributed: if N is the number of the used physical nodes, C the number of program clauses (i.e. the number of Manager processes) and K the total number of Monitoring processes, we can distinguish three solutions:

a) K=N. This is the completely centralized solution, since we have a single, global monitor responsible for each process allocation. The advantage is that decisions about allocation are taken on the basis of the global load in the system, but the intensive communications traffic to/from the monitor causes performance degradation.

b) K<C. The physical network can be partitioned in domains each one associated with a clause. For each domain, a monitor process handles the dynamic allocation of ROSE generated processes. This is a more
The evaluation of the different monitoring schemes is included in the
Machine, for the parallel execution of ROSE programs on non-shared
of multiple-headed clauses in a very incremental and parallel way.

The parallel model of the logic programming language ROSE is more suitable
This new model of communication implies that the AND/OR process tree
peculiar of parallel logic programming models becomes a graph. The new
objectives of our future works. However, we are aware that allocation is
strictly related to the user program. For this reason, we plan to investigate
the application of program analysis techniques such as abstract
implementation to obtain better allocation policies. Moreover, we are working
on implementing the overall ROSE language taking into account also the
sequential part and its integration with the parallel one.

Acknowledgements

This work has been partially supported by the "Progetto Finalizzato Sistemi
Informatici e calcolo Parallelo" of C.N.R. under grants n. 89004269 and n.
8900121.

References

[1] E. Y. Shapiro, "A Subset of Concurrent Prolog and its Interpreter", in


V2", Proc. 6th Int. Conf. on Logic Programming, Lisbon (P), (G.

GHC", Proc. 7th Int. Conf. on Logic Programming, Jerusalem.

Axiomatics and Formal Semantics of Generalized Horn Clauses",


[7] A. Brogi, "AND-Parallelism without Shared Variables", in Proc.
7th Int. Conf. on Logic Programming, Jerusalem, (D.H.D. Warren

in logic programs", in New Generation Computing, n.3, pp. 43-70,
1985.


[10] S.O. Nyström: "Control Structures for Guarded Horn Clauses", in
Proc. 5th Int. Conf. on Logic Programming, Seattle, (R.A.Kowalski,


[13] J.S.Conery: "Binding Environments for Parallel Logic Programs in
Non-Shared Memory Multiprocessor", Proc. IEEE Symposium on

Independent Binding Environment for AND/OR Parallel Execution
of Logic Programs", Proc. 5th Int. Conf. on Logic Programming,

Execution of Logic Programs", in Proc. 5th International
Conference on Logic Programming, Seattle, (R.A.Kowalski,


[19] P.O. Besco, et al., "Logic and Functional Programming on Distributed
memory Architectures", Proc. 7th Int. Conf. on Logic Programming,

system", in Proc. 1st North-American Conf. on Logic Programming,

[21] M.V. Hemmergildo, K.J. Greene, &-Prolog and its Performance:
Exploiting Independent And-Parallelism", in Proc. 5th Int. Conf.
on Logic Programming, Jerusalem, (D.H.D. Warren and Peter