A Static Implementation of the Basic Andorra Model for Distributed Memory Systems[⋆]

V. Benjumea J.M. Troya
{benjumea,troya}@lcc.uma.es

Dpt. Lenguajes y Ciencias Computación
Universidad de Málaga
Spain

Abstract. The paper shows an implementation model to support the execution of the Basic Andorra Model on distributed memory systems. A model for exploiting dependent AND parallelism in Prolog programs on distributed systems is also proposed, as well as the way in which it has to be combined with an OR parallel model. A mechanism for dealing with unbound variables in a distributed environment is proposed. In the current implementation, the arrangement of workers in the system in order to mix both models is done in a static way at the beginning of computation.

Keywords: Distributed Systems, Parallelism, Logic Programming, Andorra Model.

1 Introduction

The exploitation of parallelism in Prolog programs has been studied for the last years. Many models have been proposed in order to exploit OR parallelism [15, 20], independent AND parallelism [7, 11], dependent AND parallelism [5, 18, 19], and combinations among them [4, 10, 12]. However, most of them are focused on shared memory systems, but only a few of them have been oriented towards distributed memory systems [2, 3, 1, 13]. Nevertheless, distributed memory systems are becoming more and more important in modern parallel systems.

The Basic Andorra model [16, 17] modifies the Prolog execution model in order to transparently exploit OR parallelism combined with dependent AND parallelism. The latter has been the basis on which committed-choice languages [5, 18, 19] have been developed, thus providing a natural way of combining Prolog with this kind of languages.

In this paper we present model to support the implementation of the Basic Andorra Model on distributed memory systems. We also propose a model for the exploitation of dependent AND parallelism in Prolog programs on distributed memory systems, on which our implementation of the Basic Andorra Model is based. The OR parallel model for distributed systems is based on the model proposed by the authors in [1], therefore, and due to space limitations, it will not be

⋆ This work was supported by the project CICYT-TIC 340/90

explained here. At this stage of development, the arrangement of workers in the combination of both models is achieved in a static way in order to exploit both kinds of parallelism. That is, processors in the distributed system are arranged in a fixed way at the beginning of computation, and they are coordinated in order to exploit both kinds of parallelism. Moreover, side-effects predicates are not supported. The system implementation is nearly finished, and we hope to present some results in the nearly future.

The problem that arises when dealing with unbound variables in parallel execution is addressed. Moreover, a mechanism for dealing with such an unbound variables in parallel computations, and more precisely in distributed environments is also proposed. This mechanism is defined as an extension to the Basic Andorra execution model (it is embedded in it), besides an extension to the Foster's *distributed unification scheme* [9] in order to deal with unbound variables in distributed systems, making it more suitable to our extension of the Basic Andorra execution model.

The paper is organized as follows. Section 2 shows the model on which the combination of both kinds of parallelism is made possible. Section 3 explains the model on which we have based the exploitation of dependent AND parallelism. The model is divided into a model for supporting determinacy detection and goal suspension in sequential Prolog, and a model for parallelizing suspended goals in a distributed environment. Section 4 presents how the AND parallel model and the OR parallel model can be combined. Section 5 analyzes the proposed model.

2 The Basic Andorra Model

The Basic Andorra model [16, 17] arises as an alternative to the Prolog execution model in order to be able to exploit dependent AND parallelism in addition to the intrinsic OR parallelism in Prolog programs. OR parallelism arises in a natural way in the Prolog execution model, since a Prolog program normally defines several alternative paths to be traversed, and during the traversing of a determinate path, some information will be recorded in order to be able to traverse different paths at later stages. It is the traversing of the different alternative paths defined by a Prolog program that makes the exploitation of OR parallelism possible by allowing different *workers*[1] to traverse different paths in parallel.

However, exploitation of dependent AND parallelism is a lot more difficult to manage in Prolog execution. Thus, the Basic Andorra model was proposed to solve this drawback by modifying the Prolog execution order. The Basic Andorra execution model defines two different phases in which Prolog goals are executed. In the *deterministic phase*, execution of non-deterministic goals[2] is suspended and only deterministic goals[3] are executed. It can happen that due

[1] A worker is an entity able to achieve a task.

[2] A non-deterministic goal is such that matches with several clauses.

[3] A deterministic goal is such that matches just with one clause.

to the execution of deterministic goals, some goals that were suspended because of their non-determinacy, now become deterministic, and so, their execution will be resumed within this phase. When all of the deterministic goals have been executed, i.e. each suspended goal is non-deterministic, a non-deterministic goal will be selected, according to the Prolog execution order, to be reduced within the *non-deterministic phase*. It will be started a new *deterministic phase* where new deterministic goals will be reduced, and so on. The alternating of both phases will go on until the end of computation when all goals have been reduced.

It is during the *deterministic phase* when dependent AND parallelism can be exploited, since executed goals are deterministic and just one flow of data exists. Therefore, those goals can be executed in any order, and even in parallel, being synchronized by the flow of data.

During the *non-deterministic phase*, a non-deterministic goal will be executed, which will create a choice-point that will make it possible to restart the computation state at that point in order to follow a different alternative path at a later stage. These choice-points are the key points in the exploitation of OR parallelism, since they define different alternative paths that can be traversed in parallel by different workers.

Execution of deterministic goals earlier than non-deterministic ones brings about a possible reduction of the search space with respect to the Prolog left-to-right execution order, since some failures can be detected earlier, pruning branches in the search space, this feature is an important speed-up factor for suitable Prolog programs.

3 Dependent AND Parallelism Exploitation of Prolog Programs in Distributed Systems

In this section we will look at some general concepts related to the exploitation of *dependent AND parallelism* in distributed systems. Thus, *dependent AND parallelism* is characterized by small grain task creation, which is not a great drawback in shared memory systems. However, in distributed systems it becomes an important source of inefficiency, due principally to the high cost of remote task creation in relation to its small lifetime. This fact justifies one of our main design goals, to increase the parallel task granularity.

The other main design goal consists on achieving a suitable and efficient sharing of work. In order to get this, not every parallel task has to be shared out, but we try to keep all workers working for as long as possible with the least amount of task sharing as possible, favoring in this way local work and therefore decreasing the accesses to remote data, which is another important source of inefficiency.

3.1 Goal Suspension in Sequential Prolog

Checking for Determinacy The Basic Andorra Model relies in the fact that deterministic goals are executed before the non-deterministic ones. Thus, the first thing we have to do is to distinguish, in a Prolog program, which goals are deterministic, and which ones are not. However, this feature, most cases, cannot be detected completely at compile-time, therefore, in order to minimize the overhead that introduces, most of analysis, as in Andorra-I [17], will be done at compile-time and the execution of small tests will be achieved at run-time. Therefore, the WAM instruction set has to be extended to support run-time checking to detect deterministic goals. Thus, during compilation of a Prolog program, in addition to the WAM code, code for "goal determinacy checking" and "guards" will be generated by the determinacy analyzer. So, during the execution of a goal, it will be taken as deterministic if the checking code proves it, otherwise the goal will be taken as non-deterministic (although it can be taken as a deterministic goal later, if proved).

Goal Suspension Goal suspension is the key point that supports the exploitation of AND parallelism in the Andorra model. In our execution model, goals are executed sequentially as in WAM, but the WAM execution model is modified in the fact that the execution of *some special goals* is postponed to a later stage. Each postponed goal, and the computation derived from it, will be executed sequentially as explained. Goal postponement can be due to two main reasons. The first one is based on the fact that the Basic Andorra Model, during the deterministic phase, only deterministic goals are executed, and non-deterministic ones are postponed, thus the execution of goals that have not been proved to be deterministic will be suspended. The second reason for goal postponement is based on the creation of a framework to allow the exploitation of AND parallelism. So, a suspended goal (and the work that derives from it) can be executed by another worker while the origin worker is still executing other goals, exploiting so AND parallelism.

A goal is suspended by storing enough information to allow its later execution. This goal, as in JAM [6], will be linked in lists from variables (if the goal was suspended because of its non-determinacy) or in lists where the goal can be taken to be executed (possibly by another worker). Moreover, to allow the execution of non-deterministic goals following the Prolog execution order, it exists a double linked list that holds each suspended goal in the right order. Thus, if a goal is reduced and, as result of its execution, some goals are created, these ones will replace the original one in the double linked list. When a non-deterministic goal is going to be executed, it will be taken from this double linked list, following so the Prolog execution order.

This suspension of goals can be seen as the creation of work, which will be carried out later at a convenient stage (or it could be shared with another *worker* to exploit AND parallelism). Thus, the resolution of a Prolog program can be roughly seen as the execution of an initial sequential work that can create more sequential work (deterministic and non-deterministic), which will be executed when the current work has been carried out. Part of the created work will be selected for execution, which could create more work, and so on.

The computation will finish when the current work has been carried out and no work remains to be executed.

However, in general, the suspension of goals degrades the performance of the efficient sequential WAM execution model (an exception to the rule consists in the reduction of the search space, which will be described below), therefore only goals suitable for AND parallel exploitation should be suspended. Due to the degradation of the system performance, the work division has to be done carefully trying to keep as much of the original sequential WAM execution of goals as possible, increasing so the task granularity.

Creating Work The basic Andorra model support relies in the suspension of non-deterministic goals and their later execution as it was seen above. However, if we want to improve the performance of our system by sharing the computation between system workers, we need a way to divide this computation in parts suitable for sharing. Thus, suspension of goals (deterministic and non-deterministic) is the used mechanism to divide the work in smaller parts suitable for sharing. However, our system is focused on exploiting parallelism in distributed memory systems, where remote task creation is expensive, therefore remote task lifetime have to be big enough to make its sharing worthy, so we cannot make a correspondence between a goal and a task as JAM [6] can, but a *"task will be rooted in a goal, and it will comprise all computation deriving from that goal"*, unless that such a work is divided. Thus, a task is a partial resolution, following the sequential WAM execution model, of a goal. We say partial because this sequential execution can create more tasks by suspending some goals. This creation of tasks can be led by program analysis [14] or by user annotations.

The Execution Model In order to support the Basic Andorra Model, the WAM sequential execution model has been modified to have two main phases, which alternate each other until the end of computation. The first phase is the *deterministic phase*, which follows a sequential execution, including determinacy checking code, of involved goals (a task). If during this phase a goal cannot be proved to be deterministic, it will be suspended. Moreover, a goal can be suspended, too, as a consequence of a work creation instruction. When the current task has finished, another task will be selected for sequential execution. This selection will be achieved within a sub-phase where a deterministic task (a deterministic goal points to a deterministic task) will be searched. Therefore, within this sub-phase, deterministic checking code of goals is executed to know whether some available goal is deterministic. Found non-deterministic goals are suspended again, the first found deterministic goal is selected to be the current task and its sequential execution is started within the *deterministic phase*. When the current task execution has finished and no deterministic task can be found, which means that there is no deterministic goal available in the system, then the system falls into the second phase (the *non-deterministic one*), where a non-deterministic goal, according to the Prolog execution order, will be selected from the double linked list for non-deterministic execution. If there is no non-deterministic goal available when it has to be selected, then a solution for the initial query will have been found. When a non-deterministic goal is executed in the non-deterministic phase, it creates the correspondent *choice-point* and the current state changes to the *deterministic phase*.

Backtracking Support This section will explain the extensions to the WAM backtracking mechanism in order to support the suspension of goals and the two execution phases in sequential Prolog.

The main change is with respect to the logic variable unique assignation property, which externally remains unaltered, but internally, variables, goal record fields, etc. now hold many different values depending on the computation path. The WAM trail mechanism is based on the unique assignation property, where it is only necessary to keep the addresses of conditionally bound variables[4] in order to unbind them when backtracking occurs. In our model, whenever an object is bound for the first time, the trail mechanism acts like in WAM, but if the object is bound again to new values, the trail mechanism is modified by storing the object old value and its address in a new area (*Trail_Values*). Thus to restore the object values to a determinate state of computation, the untrailing mechanism is modified to use this information.

Dealing with Unbound Variables As it was explained above, when the determinacy of a goal depends on the value of some variables, and any of them is unbound, then the goal is suspended. The same behavior can be applied to built-in predicates. However, sometimes, determinacy of a goal depends on the fact that a variable is unbound, so let us take a look at the following code inspired by [21]:

```
1.1) send_msg(M, Msg) :- var(M), !, M = [Msg|_].
1.2) send_msg([_|M], Msg) :- send_msg(M, Msg).

2.1) get_word(W, [], []) :- !, W = [].
2.2) get_word(W, [X|Rest], Rest) :- X == ' ', !, W = [].
2.3) get_word([X|W], [X|T], Rest) :- X \== ' ', !,
                                   get_word(W, T, Rest).
```

In this example, the *'send_msg/2'* predicate will be always deterministic, thus if the first argument is “unbound” the first clause will be selected, and otherwise the second one. With respect to the *'get_word/3'* predicate, if the second argument is unbound, its determinacy cannot be proved, but if it is bound, then the predicate will be deterministic, choosing the first clause if it is bound to the empty list, the second one if it is bound to a list which head is bound to the space (' ') and the third one if it is bound to a list which head is “unbound” or is bound to anything different from space.

[4] A variable is conditionally bound if since it is created until it gets bound to a value, some choice-point has been created.

In such cases, if the execution order is fixed (like in Prolog), the program will have a known behavior, but if the execution order is not fixed (like in Andorra during the *deterministic phase*) the program will have an unpredictable and non-deterministic behavior. Moreover, in parallel executions, results can be worse. Thus, for example, the following query

```
... send_msg(M, msg1), send_msg(M, msg2) ...
```

would raise different results depending on the execution order of the calls

```
M = [msg1, msg2 | _ ] or M = [msg1, msg2 | _ ]
```

even though the predicate is deterministic. Even worse, it would fail if both calls are executed in parallel.

This problem seems reasonable since if a clause has to be selected depending on the value of a variable, or even if the variable is unbound, and there is not a strict execution order, the time when the variable gets bound will affect to the computation. Thus, this execution model has a non-deterministic behavior in case of dealing with unbound variables.

Andorra-I [17] overcomes this problem by mean of its preprocessor, which makes a preliminary analysis of the Prolog program through *abstract interpretation*, and establishes a sequential ordering in *sensitive* and related predicates.

We propose to extend the execution model in order to define a deterministic behavior even in such cases. Thus, when the determinacy of a goal depends on a variable that is unbound, the goal will be taken as *non-deterministic* (while the variable remains unbound) even if the goal is deterministic for the case of the unbound variable. Therefore its execution will be delayed up to the *non-deterministic phase*, sequencing so the non-deterministic access to unbound variables. Thus if any deterministic goal bind the variable to any value, the non-deterministic goal will become now deterministic. However, if the variable remains unbound, then the goal will be executed in the *non-deterministic phase* and will select the clause related to the unbound variable. The model can be seen as follows: if the determinacy of a goal depends on a variable that is unbound, the execution of the goal will be delayed until no other goal could bind it. If any goal binds the variable, then its value will be taken to select the deterministic clause (if exists), and if no one binds the variable, then it will be taken as unbound for the delayed goal.

The same behavior can be applied to any built-in predicate that dealt with unbound variables, it will be delayed until the variable became bound or up to the *non-deterministic phase.*

This extension to the execution model guarantees a deterministic behavior in the execution of an Andorra program, and it will conform the execution order of Prolog in most cases.

3.2 Exploiting Dependent AND Parallelism in Prolog

Exploitation of dependent AND parallelism is based on the goal suspension mechanism seen above, which allows the division of the computation into smaller parts (tasks) suitable for sharing between system workers, exploiting so AND parallelism. However, it is important to take into account that our system is oriented to run on distributed memory machines, therefore task division and its sharing have to follow several guidelines to take advantage of distributed execution.

In distributed systems, remote task creation is an expensive operation. For this reason, it is only worthwhile if the task execution time is high enough with respect to its remote creation cost. Most of systems that exploit dependent AND parallelism follow an execution model similar to JAM [6], where in general each goal corresponds to a process, in this way having a big amount of little processes, exploiting so small grain parallelism. Our model follows a different approach in order to exploit medium/coarse grain parallelism, thus a task is executed internally following the modified WAM sequential execution model explained above (which can create more tasks), and the creation of tasks will be controlled by the *work creation instructions*. So, in our model, a task will not only reduce a goal, but it will execute sequentially all the work derived from such a goal, and possibly it will create more tasks. In this way, the granularity of tasks is increased. The work creation instructions will be generated by program analysis [14], user annotations, etc. and it can be used in conjunction with *extended instructions* for a higher control of parallel tasks.

Our system can be seen as a *multisequential* AND parallel machine made up of workers, each one executing a task in a sequential way similar to WAM execution model, and possibly with some tasks waiting to be executed.

Sharing Parallel Work In this section we will see how tasks are shared between systems workers in order to exploit dependent AND parallelism, and how these workers synchronize each other to share work in a suitable way.

The first thing to take into account is that sometimes it could be convenient to break the sequential execution flow in order to postpone the execution of a task or to anticipate the creation of another task, etc., and it could happen that the new created task is not suitable for sharing with other system workers. Therefore our model divides tasks into two main classes, *parallelizable tasks*, which are suitable for sharing between workers in a distributed system, and *local tasks* which are not convenient for sharing, but they are useful for breaking the sequential execution flow.

Another important thing to take into account is that the execution of a task normally will need data produced by other tasks and it will produce data for other tasks[5]. This behavior is quite different from the exploitation of OR parallelism, where tasks are coarse grained and interaction between them is at creation task time. Thus, one of our more important design goals is to get an efficient sharing of AND work, therefore creating tasks suitable for parallelizing (medium/coarse grained task) is not sufficient, but such tasks have to be shared in an efficient way between all system workers. To achieve this goal, we try to keep workers working for as long as possible, but minimizing the amount of task

[5] We are exploiting *dependent AND parallelism*.

exportation[6]. In this way we get to reduce remote task creation costs and remote data access costs, but we still exploit dependent AND parallelism in an efficient way.

At the time of designing a parallel system on a distributed architecture, abstraction and independence with respect to the underlying architecture is the right approach and has numerous advantages. However if we want to get the best possible performance from the underlying architecture, we need to know some additional information to help us in our work. Thus, in a distributed environment, communication costs vary from some processors to others (depending on the network topology), and this information can be useful, keeping abstraction and independence, for our purposes. Therefore, in our system each worker is surrounded by *levels of neighborhood*, where each level will hold workers that have the same communication costs (Fig. 1). This means that workers have to be ordered by levels depending on their communication costs. Thus this information will be used by our system during the sharing of work to select the most suitable workers with which to do it.

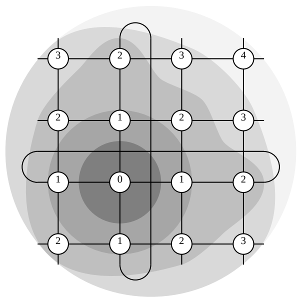

Fig. 1. Levels of neighborhood

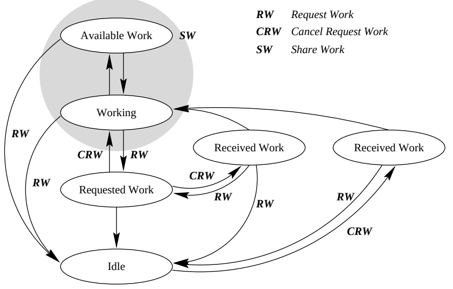

Fig. 2. State machine

The sharing of tasks is controlled by the state machine in fig. 2 which synchronizes the operation between involved workers. Normally a worker will be in the *"working"* state, which means that it has got enough work to be running but it has not got enough parallel work to share, or in the *"available work"* state, which means that it has got enough parallel work for sharing. When a worker is in this latter state and knows that another worker needs work, it will share a part of its work with it. Whenever the amount of work of a worker is below a determinate level, it will request work from the neighbor workers and will move to the *"requested work"* state. If a worker runs out of work, it will move to the *"idle state"* and it will have to wait to get some work from neighbor workers. When a worker receives some work, it will move to the *"received work"* states. Moving between states is governed by thresholds that mark the amount of work

[6] Task exportation follows a lazy approach.

needed to stay at a determinate state, and the amount of shared work is defined by the ratio of sharing.

A state machine is defined for each level of neighborhood, and only neighbor workers belonging to such a level will be involved in the sharing of work, getting so a finer control depending on its communication costs. Thus, each level of neighborhood will have defined suitable thresholds and ratios to produce an efficient sharing of work. The choosing of suitable thresholds and ratios can be done by mean of program analysis.

Thus, the sharing of work can be seen as follows: when the amount of work in a worker is under a determinate threshold (the worker gets into the *"requested work"* or *"idle"* state), it requests work from the neighbor workers from the distance[7] specified by the state machine. When a worker has got enough work to share (it is in the *"available work"* state), it will select a worker that has requested some work within the distance specified by the state machine. Then it selects the most suitable piece of work (the amount of tasks specified for the ratio of sharing for that distance) to send it to the worker that requested it.

The way in which the tasks are selected for sharing is, in order of precedence, as follows:

1. A task imported from the same worker which is now requesting work.
2. A task imported from another worker.
3. A parallelizable task created by the worker that is going to share its work.

It is important to note that this behavior tries to keep as many tasks as possible located at workers where they were created, with the minimum amount of task exportation, since sharing of work is achieved in a lazy way by demand when the amount of work is below a specified threshold.

Whenever a task is exported, it is marked as such and records a remote reference to the created remote task. On the other hand, when a task is imported, it is marked as such and records a remote reference to the original task, moreover, a message is sent in order to set up the reference from the exported original goal. It has been seen that imported tasks can be re-exported to other workers, so it is necessary to keep a reference count in each task in order to synchronize the reference messages to the original goal. Imported goals are linked in the lists of goals available for execution.

We have seen that when a goal is executed, it is replaced in the double linked list by its descendant suspended goals. When a goal is exported, it will no be removed from this list, since it will be used to control the Prolog ordering in the selection of a goal during the non-deterministic phase. Imported goals are not linked in the worker double linked list (they will be executed in the non-deterministic phase through the original exported goal). When an imported goal is executed, its descendants will not replace it, but they will be linked in a double linked list rooted in the father imported goal. In fig. 3 can be seen how the Prolog ordering is maintained. Thus, there will be a *"main double linked list"*

[7] The level of neighborhood.

and one of them for each imported goal. In this way, and using the references between exported and imported goals, a *"global double linked list"* is maintained distributed in the system.

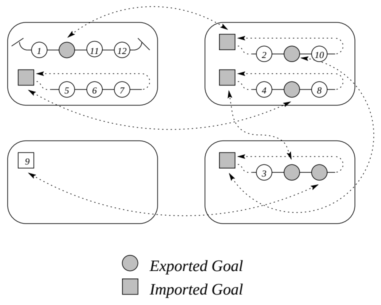

Fig. 3. Distributed double linked list

Sharing Data between Parallel Tasks. Extended Distributed Unification This section is based on Foster's distributed unification model for committed choice languages [9], which has been extended to support the mechanism for dealing with unbound variables in a distributed environment. Communication between tasks is achieved by mean of shared logical objects, which are stored in WAM memory areas. These shared objects will be distributed, or in some cases replicated, between the system workers (each worker holds its own memory areas). Thus, communication of tasks will be achieved by accessing a variable to unify it with some value (variables keep their property of unique assignment), or by accessing an object in order to get its value. So in this way, the tasks within a worker communicate with each other by means of objects residing in the same shared WAM memory areas. However, a more sophisticated mechanism is needed in the case of tasks residing in different workers, which need to access remote objects in order to achieve reading and unification operations.

The way in which distributed data are related each another is by means of *remote references*. These are a mean of relating a variable residing in a worker local memory to an object residing in a remote worker memory. This is done simply by referring the worker where the object resides and the object with which the remote reference is related.

Whenever a task needs to access to a variable value, if the variable is bound to some object, it just takes it. If the variable is unbound and its value is necessary to know whether a goal is deterministic or not, the goal will be suspended in a suspension list attached to the variable. If the variable value is necessary for the execution of a built-in predicate, this will be suspended in the variable suspension list. It can happen that the variable was a remote reference to an object in a remote worker. In such cases, a message to get the value from the remote object will be sent to the worker that owns the object. Meanwhile, the goal will be suspended in suspension lists attached to variables on which its determinacy depends.

When a worker receives the message asking for the value of an object, if the object is available, it just sends it to the worker that requested it. If the object is a reference to another remote object, the requesting message is forwarded up to get to the worker that owns that object. However if the object is an unbound variable, it will attach a note to the suspension list of the variable, in such a way that when the variable becomes bound, its value will be sent to the worker that requested it.

The other way of accessing a variable is its unification with another object. If both objects are local to the task that wants to unify them, the operation is achieved as in WAM. If the unification involves a local object with a remote one, the local object is sent to the worker where the remote object resides in order to unify the local object with the received one. When both objects are remote, it will send a message to the worker where one object resides to achieve the remote unification operation with the other worker as seen above.

It is important to note that remote references only can be done in one direction, in order to avoid remote circular references. Thus, in our system, worker identifiers are ordered, and only remote references from higher to lower worker identifier are allowed. Thus, operations explained above and remote object creation must be achieved according to this rule.

There are, however, several built-in predicates[8] which achieve *non-strict tests*, that is, they can sometimes proceed even though their input arguments are unbound [9]. However Foster's *non-strict read algorithms* are oriented to *committed choice languages*, which have a very poor treatment of unbound variables. Therefore, in our model, the *non-strict read algorithm* is modified in order to support the *Basic Andorra Model* extended with the *unbound variable treatment* on distributed systems.

As it was said above, goals which determinacy depended on unbound variables were taken as non-deterministic ones, and so, executed at *non-deterministic phase*. Foster's distributed unification algorithm guarantees that if the value of a variable is needed for any goal in the distributed system, and the variable become bound before the *non-deterministic phase* start, then the value of the variable will get to the goals that need it before the beginning of the *non-deterministic phase*. What we have to get by extending the distributed unification mechanism is to guarantee that whenever a goal needs the *"last"* reference of an "unbound" variable[9], such a last reference must be available at the start of the *non-deterministic phase*, since execution of such goals will be delayed up to then. It is important to note that not every non-strict predicate (such as

[8] '==/2', '\==/2', 'var/1', 'nonvar/1', etc.

[9] Some built-in predicates such as '==/2' and '\==/2' compare terms even if they are unbound. If such predicates appear within the determinacy code or guards of a Prolog predicate, then the Prolog predicate needs their values in the same way.

'var/1', etc.) needs the last reference of the unbound argument, therefore the next discussion will not be applied to them.

The way in which the right (and last) reference for an unbound variable is guaranteed to be available when needed at *non-deterministic phase* can be roughly seen as follows:

> When a goal needs the *last-reference* from a variable for a non-strict test, a *note*, holding information about the variable whose last-reference is needed, will be attached to the current last-reference, and whenever the last-reference gets bound to another variable, the goal is notified[10] and the *note* is forwarded to the new last-reference. If the last-reference becomes bound to a value, the goal is notified and the note is discarded.

From Deterministic to Non-Deterministic Phase. Detection and Control In this section we will see how the change from *deterministic* to *non-deterministic* phase is controlled by the system workers in order to maintain the Prolog execution order for non-deterministic goals.

As it was seen, the alternating of phases will be done until the end of computation. In the AND parallel model, this alternating will have to be achieved in a synchronized way. At a determinate moment, all workers will be in the *deterministic phase* executing tasks in parallel. When there is no available task in the whole system, i.e. every worker in the system is in the "idle" state, then in a synchronized way all workers change to the *non-deterministic* phase. One of them will execute a non-deterministic goal[11], and will start a new *deterministic* phase by executing the task generated by the non-deterministic goal. Every worker will now change to the same phase and will execute their tasks[12]. Thus, we have seen that it is one worker that starts a new *deterministic* phase, we will call it the *initial worker* for each phase.

Let us go to the start of the Prolog computation. There will be an *initial worker* that will start the *deterministic phase* by executing the *initial Prolog query*. During the execution of the initial sequential task, some tasks could have been created, from which, when it gets to the *non-deterministic phase*, the deterministic ones will have been executed, and the non-deterministic ones will remain waiting for their execution following the Prolog execution order. Processing will go on with phase alternating as explained above up to the point when the selected goal for non-deterministic execution was an exported task. Then a message has to be sent to the worker that owns that task in order to start the non-deterministic execution of that goal or, in the case of it had already been executed, the first non-deterministic goal created by such a task. The worker that sent the message will wait for the beginning of the new *deterministic phase* and non-deterministic goals in such a worker will be executed when all

[10] It must be taken into account that such messages can arrive in an unordered way.
[11] According to the Prolog execution order
[12] Which will have been imported from other workers or will have been awakened by mean of shared logical variables.

non-deterministic descendants from the exported goal have been executed. The worker that receives the message will execute the selected non-deterministic goal (or one of its descendants) and will start a new *deterministic phase* (being now the *initial worker* for that phase). It will return the non-deterministic control, to the worker from which it got it, when all of the imported goal descendants have been executed. However, if some of these goals are exported, the process will go on as explained, giving workers the non-deterministic control and returning it at many levels in order to keep the Prolog execution order (fig. 3).

We have seen that it is the *initial worker* that will change to the *non-deterministic phase* by selecting the next non-deterministic goal to be executed, which could have been exported. However, to start a new *non-deterministic phase*, all available deterministic goals must have been executed, i.e. there must not remain any available deterministic task in the system workers without being executed. Therefore it is the initial worker for that deterministic phase who has to detect when all workers are in the "idle" state in order to start the new *non-deterministic phase*.

It has to be taken into account that in distributed systems, not all computation is done inside the workers, but some messages can be traveling around the interconnection network. So, in order to know if a determinate worker is in the idle state, the workload is not the only important value to be taken into account, but the messages which it is waiting for have to be taken into account too. That is, a worker will move to the "idle state" if it has not got any work available to be executed and it has received an acknowledgment for every message that it has sent, in which case there will not be messages traveling around the network.

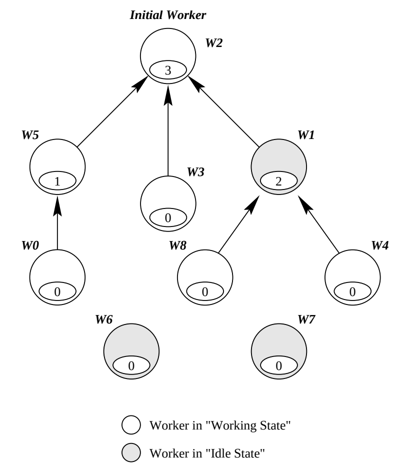

Fig. 4. Detection of the End of the Deterministic Phase

Detection of the End of the Deterministic Phase The algorithm for the *detection of the end of the deterministic phase* is based on the *diffusing computation* termination detection method proposed by Dijkstra and Scholten [8], but it has been oriented to workers instead of tasks[13]. In it, each worker is controlled by the worker which has given it work, and each worker controls the workers to which it gave work. Thus, each worker keeps the worker that controls it, and the count of workers that it controls, following a tree hierarchical structure (fig. 4). The algorithm is as follows:

- When a worker gets some deterministic tasks from other worker and it is not controlled by any worker, the control is established between both workers.
- When a worker has not got any available deterministic task to be executed (it is in "*idle* state"), and it does not control to any worker, then the control from its controller worker is removed.
- If the *initial worker* is in the "*idle* state" and does not control to any worker, that means that there is no available deterministic task in the whole system, and, therefore, the non-deterministic phase must begin.

Backtracking Support In addition to the backtracking support for sequential execution explained above, our model has to be extended in order to support backtracking in dependent AND parallelism exploitation in a distributed environment.

It is known that *choice-points* are a key piece in the WAM backtracking mechanism, and we have seen that a non-deterministic goal is executed just by one worker (the *initial worker* for a deterministic phase) and it will create a choice-point in its local memory. Then a message for creating a *dummy choice-point* is broadcasted to all system workers[14] in order to allow every worker to backtrack to that state of computation in case of *failure*. After the *initial worker* creates its choice-point and broadcasts the *dummy choice-point* creation message, the *initial worker* will go on executing its new task, however, it cannot send any remote work creation message[15] until it has received confirmation from every worker of having created their correspondent dummy choice-points. This is achieved by having a *'queue'* where such a messages will be enqueued. This synchronization is necessary for the choice-points to reflect the same global computation state.

In the same way, when the *initial worker* discards a choice-point, it broadcasts a message to every worker in the system in order to act in the same way.

[13] The main difference resides in the fact that in the original method, a task waits for termination of its children in order to finish its computation and to notify this fact to its father. In our environment, a worker can receive work from many workers and can become idle even if its children are working.

[14] Every worker is at this moment at *"idle state"*, since the change of phases is done in a synchronized way.

[15] Remote task creation, requested object value/reference or remote unification.

Synchronization as explained above is required. Moreover, in case of an early failure after the creation/destruction of a choice-point, the failure has to be synchronized with the confirmation from every worker of having created/discarded the dummy choice-point.

When a failure occurs in a worker, a failure message is broadcasted to every worker in the system. However, workers are working asynchronously most of time[16], therefore many failures can happen in different workers over the system, and many obsolete messages can be traveling over the network. Thus, in order to control this asynchrony chaos, a failure level counter is used. Whenever a failure occurs in a worker, the failure level counter is incremented. Moreover, every message is tagged with the worker's failure level count, in such a way that when a message is received, if the message failure level count is lower than the worker's current failure level count, then the message is discarded as obsolete. However if the message failure level count is higher than the worker's current failure level count, the worker has to fail so many times to get to the message failure level count, and then the message will be processed.

4 Mixing OR Parallelism with Dependent AND Parallelism

This section shows the way in which the system is arranged in order to exploit both kind of parallelism in distributed systems. It is based on the dependent AND parallelism model explained above and in the OR parallel model explained at [1] which is based on the *incremental copy mechanism* [2] for distributed environments. It will not be explained here due to space limitations.

4.1 Static Teams

The structure that governs the whole system in order to exploit both kinds of parallelism is arranged as follows: systems workers are divided into fixed size teams, each one made up of the same number of workers. Workers have identifiers internal to the team and teams have their own identifiers. The system can be seen from two points of view. From the outer level point of view, the system is made up of a set of *teams* which collaborate one another exploiting *OR parallelism* by traversing in parallel different alternative paths from the search space. From the inner level point of view, each team consists in a set of workers which collaborate one another exploiting *dependent AND parallelism* within the traversing of a search space path. This two level approach can also be applied to the levels of neighborhood defined for both models. Hence workers within a team can be organized by levels of neighborhood, and teams within the system can be organized in such a way too.

Let us look at some definitions and restrictions that system team division must fulfill.

[16] Workers are only synchronized at change of phases.

– Definitions:
  1. Let d_{ij} be the distance between worker i and worker j within a team. Then let us define the *internal distance* for a team T_t with N workers as
$$ID_t = \sum_{i=0}^{N-1} \sum_{j=0}^{i-1} d_{ij}$$
  2. Let us call the worker's *counterpart* for a team T_j to the worker which has the same identifier as it within such a team, and let D_{wij} be the distance between worker w from team T_i and its counterpart from team T_j in the system network. Then let us define the *external distance* between two teams T_i and T_j with N workers as
$$ED_{ij} = \sum_{w=0}^{N-1} D_{wij}$$

– Restrictions:
  1. The team *internal distance* must be minimum, that is, division of workers in teams which internal distance is not minimum is not allowed.
  2. The *global external distance* between all teams must be minimum, that is, team layout and the arrangement of workers within a team must be achieved in order to minimize
$$\sum_{i=0}^{NTeams-1} \sum_{j=0}^{i-1} ED_{ij}$$

  These restrictions are in order to decrease the communication costs within a team while exploiting Dependent AND parallelism and while the sharing of work in OR parallel exploitation respectively.

4.2 Mixing both Models

Now we will see how the proposed models have to be modified in order to adapt themselves to the system arranged in fixed size teams.

The *Dependent AND parallel model* is lightly modified in order to allow the mixing of both models. In previous discussion, all workers in the system were involved in the same AND parallel computation. In order to exploit both kinds of parallelism, only workers within a team will be related in the same AND parallel computation, and workers from different teams will be involved in different computations. The model is modified as follows: a team is a subset of workers from the whole system, and worker identifiers are now relative to the subset, hence the model must reflect the relative worker identifiers and the subset of workers as a whole AND parallel system.

The *OR parallel model*, however needs deeper modifications. It has to be modified to support the teams, instead of workers, as the core to exploit the OR parallelism. Now a team plays the same role as a single worker in the OR model. These teams are made up of small pieces that will have to be coordinated in order to achieve the sharing of OR work in the correct way (now the OR work is not a monolithic piece, but it is made up of small parts distributed among the workers belonging to a team).

Therefore, it is the sharing of OR work, and the corresponding coordination of workers within a team and inter-teams on which our attention must be focused.

The way in which the sharing of work was synchronized in the *pure OR parallel model* can roughly be seen as follows [1]:

- When a worker ran out of work, it sent to its neighbors a request for work, and moved to the "idle state".
- When a worker had got enough work to share, it selected the most suitable idle worker with which to share part of its work.
- When an idle worker was selected for sharing work, it chose the most suitable work for sharing and rejected the other ones.
- If *overlapping*[17] occurs before the commitment of the work[18], then the sharing of work is aborted.

This simplified protocol to synchronize two workers for sharing OR work becomes more complex in the case of the mixed models, since now two teams (made up of several workers running most of time asynchronously) have to be synchronized. It can roughly be seen as follows:

> Within a team, there will be a distinguished worker, the *"main worker"*, which will be a fixed one for each team. If a team has enough work to share, this worker will deal with the detection of work and the selection of the idle team with which to share out such work. When the work to share has been detected, another worker will deal with the control of the sharing of work and overlapping, we will call it the *"work controller worker"*, and it will be the worker that created the *"real choice-point"* just over the work to be shared.
>
> If a team is in the "*idle* state", the *"main worker"* will select the most suitable work between the offered ones, and will reject the others. The *"work controller worker"* will be notified of the work request/rejection, the work will be shared (copied) between every worker and its counterpart from both teams, and the *"work controller worker"* will be again notified of the end of such an operation.
>
> If overlapping occurs before the work has been committed, then the sharing of work will be aborted, but if it occurs after such commitment and before the end of the sharing of work, then the overlapping will be blocked until its end.

[17] Overlapping occurs when after several failures the worker that had got work to share has to backtrack to the area selected for sharing.

[18] Work is committed when the worker that owns the work receives the confirmation for sharing the work before overlapping occurs.

5 Conclusion and Future Work

We have presented a model for implementing the Basic Andorra Model on distributed systems whose implementation is nearly finished. The model is based on a model for exploiting OR parallelism on distributed systems presented in an earlier paper and in a model for exploiting Dependent AND parallelism on distributed systems explained above. The latter is based on a goal suspension mechanism for WAM with granularity control support embedded in the execution model. Moreover, levels of neighborhood and a state machine for each level are defined in order to get a fine control over the sharing of work. A mechanism for dealing with unbound variables in the distributed Basic Andorra Model has been also proposed, which is an extension to the Foster's *distributed unification mechanism* for committed choice languages. A scheme to control the change of phases maintaining the Prolog ordering and backtracking in distributed environments is proposed. A model for mixing both kinds of parallelism has been proposed, however at current stage, workers are arranged in a fixed way, being our next step to extend it using a flexible scheduler.

References

1.
2. K. A. M. Ali and R. Karlsson. The Muse Or-parallel Prolog model and its performance. In *North American Conference on Logic Programming*, pages 757–776, 1990.
3. L. Araujo and J. J. Ruz. PDP: Prolog Distributed Processor for Independent-AND/OR parallel execution of Prolog. In *Logic Programming: Proceedings of the 11th International Conference*, pages 142–156. MIT Press, 1994.
4. R. Bahgat. *Pandora: Non-Deterministic Parallel Logic Programming.* PhD thesis, Imperial College. London, February 1991.
5. K. Clark and S. Gregory. Parlog: Parallel programming in logic. *ACM Transactions on Programming Languajes and Systems*, 8(1):1–49, January 1986.
6. J. Crammond. The abstract machine and implementation of parallel Parlog. Technical report, Imperial College. London, June 1990.
7. D. DeGroot. Restricted AND parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 471–478. ICOT, 1984.
8. E. W. Dijkstra and C. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.
9. I. Foster. Parallel implementation of Parlog. In *International Conference on Parallel Processing.* University Park, 1988.
10. G. Gupta and M. Hermenegildo. ACE: And/Or-parallel copying-based execution of logic programs. In A. Beaumont and G. Gupta, editors, *ICLP'91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, volume 569 of *Lecture Notes in Computer Science*, pages 146–158. Springer-Verlag, June 1991.
11. M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, University of Texas, August 1986.
12. S. Janson and S. Haridi. Programming paradigms of the Andorra Kernel Language. In *Proceedings of the International Logic Programming Symposium*, pages 167–186. MIT press, October 1991.

13. H. F. Leung. *Distributed Constraint Logic Programming.* PhD thesis, Imperial College. London, November 1991.
14. P. López, M. Hermenegildo, and S. Debray. Towards granularity based control of parallelism in logic programs. In *Proceedings of PASCO'94*, 1994.
15. E. Lusk, D. Warren, S. Haridi, and et al. The Aurora Or-parallel Prolog system. Technical Report 07, University of Bristol, 1990.
16. V. Santos Costa, D. Warren, and R. Yang. The Andorra-I engine: A parallel implementation of the Basic Andorra model. In *Logic Programming: Proceedings of the 8th International Conference*. MIT Press, 1991.
17. V. Santos Costa, D. Warren, and R. Yang. The Andorra-I preprocessor: Supporting full Prolog on the Basic Andorra model. In *Logic Programming: Proceedings of the 8th International Conference*. MIT Press, 1991.
18. E. Shapiro. Concurrent Prolog: a progress report. *IEEE Computer*, 19:44–58, 1986.
19. K. Ueda. Guarded horn clauses. Technical Report 102, ICOT, Tokyo, 1986.
20. D. Warren. Or-parallel execution models of Prolog. In *TAPSOFT*, volume 250 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.
21. R. Yang. Solving simple substitution ciphers in Andorra-I. In G. Levi and M. Martelli, editors, *Logic Programming: Proceedings of the 6th International Conference*, pages 113–128. MIT Press, June 1989.