An OR Parallel Prolog Model for Distributed Memory Systems§

V. Benjumea, J.M. Troya
Dpt. Lenguajes y Ciencias de la Computacion
Universidad de Malaga
Spain
benjumea@ctima.uma.es troya@ctima.uma.es

Abstract

This paper shows a multisequential model to exploit OR parallelism on distributed memory systems. It presents an implementation of the incremental copy mechanism oriented to distributed systems and a novel distributed scheduler is also proposed. WAM modifications to support the proposed model are very simple. The system has been implemented on a 16 processor multicomputer based on transputers. It has been obtained very good performance results with an overhead around 6% and a speed-up comparable to the most known multisequential models for shared memory systems.

1. Introduction

OR parallelism has been widely studied since the first works achieved by Ciepielewski, Haridi [3] and others. Several approaches have been proposed to exploit OR parallelism in Prolog programs. The most significant are: the process tree models [4], where each node in the search tree is associated with a process which tries to solve it, data flow models [5,10], where the search tree is represented as a set of processes synchronized by the data flow, and multisequential models [8,9,6,1,2]. The latter are based on having multiple Prolog engines that work like sequential ones combined with a method to keep the multiple variable bindings that different Prolog engines could achieve, and with a method to share the work between the system workers[1]. The multisequential models seem to be the best approach, since they can take advantage of the very efficient sequential Prolog compilers based on WAM [7].

Several systems based on multisequential models have been proposed. The most known are the Aurora Prolog system [6] on shared memory multiprocessors, and the Muse Prolog system [1] on a multiprocessor machine with distributed and shared memory. Both of them have very good performance results. But in the scope of *pure* distributed memory multiprocessor machines there are very few multisequential systems. However multisequential models seem very suitable for this kind of architecture since they can exploit coarse grain parallelism, which implies large parallel tasks with very low interaction between them.

We have implemented a multisequential Prolog system running on distributed memory multiprocessor machines (a 16 Transputer network) and obtained very good results. The extra overhead introduced by the multisequential implementation with respect to the sequential one is very low, around 6%, and there is an almost linear speed-up for problems with coarse grain parallelism. It is based on a *distributed scheduler* that allows the work to be shared between the workers in the network, and on an *incremental copy mechanism*. The distributed scheduler is particularly important

---

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

[1] A *worker* represents a process or processor in the same sense as it does in Aurora [6] and Muse [1].

in a distributed implementation, since the communication cost is very dependent on the strategy of distributing the work between the workers. The distributed scheduler proposed defines levels of neighbourhood. It favours the cooperation between the nearest processors, establishing different task granularity depending on the distance between processors. The incremental copy idea was introduced by the Muse model in order to reduce the overhead generated by copying the worker's state to an idle one in a non-shared stack scheme during the sharing of the work. The idea is based on the fact that the idle worker could have already traversed a part of the path[2] to the work, thus it does not need to copy this part of the stacks. This idea has been oriented, in our model, to pure distributed memory machines, differing in some aspects from that proposed by Muse. The current implementation is based on a simple WAM as it was defined by D.H.D. Warren [7] without optimizations and without side-effect predicates being supported.

The paper is organized as follows. Section 2 presents a general overview of the model, defines some design considerations, describes the incremental copy mechanism, the distributed scheduler and the extensions to the WAM to support the proposed model. Section 3 shows some performance results, discusses the proposed model and compares it with others. The paper ends with a conclusion in section 4.

2. The Distributed Multisequential Model

During the traversing of a path by a worker, some work could be created. If this work becomes shareable, an idle worker, if exists, will be selected and the work will be shared with it. It is in this way how a problem is divided and solved by the system workers.

This rough scheme shows three main points to be considered. One of them is to detect when it is convenient to share the created work. Another one is how to share the work in an efficient manner. The latter important point is how these workers communicate and synchronize with each other to share the work between all of them, and how the most suitable idle worker is selected to share a part of the available work. The solving of each one of these aspects in our model is presented in the three following subparragraphs. The fourth is dedicated to the extensions made to the WAM in order to support the model.

2.1. Sharing Quantum of Work

We have been talking about shareable work, but we haven't specified what that term means. In distributed environments, where task-switching is especially expensive, it is very important to control the task granularity, as much in the worker that owns the work, as in the piece of work given to the idle worker. Thus, work (in the sense of useful work suitable to exploit OR parallelism, i.e., some kind of choicepoints) created by a worker becomes shareable when the amount of such work is higher than a determinate threshold.

In shared memory systems, different branches belonging to one choicepoint can be shared [6,1], that is, the same choicepoint is shared by several workers, which can backtrack to follow the different alternatives. However in distributed systems this type of sharing would require many messages to synchronize all workers sharing different branches from the same choicepoints. Thus, this approach seems to be very inefficient in a distributed environment. Another approach could be to share a *quantum* of work

---

[2] We will call *path* to a direct way from the root to a leaf or up to a specified node.

as a number of whole choicepoints. Thus for example, if a worker has four alive[3] choicepoints, it can give the work corresponding to a whole choicepoint to another worker. This choicepoint will then become dead[4] from the point of view of the first worker. Therefore, apart from the thresholds, the ratio of shared quantum with respect to the created work must be specified. This approach seems to be more efficient for distributed systems since synchronization in this case is very simple, as shown below.

The next decision is whether a worker gives the top-most choicepoints or the bottom-most ones. The answer seems very clear, since if the given choicepoints are the top-most ones, the overhead introduced by the memory copying operation is lower due to the lower memory area size to be copied. To reduce the frequency of task switching (and therefore of memory copying) it is very important to choose adequate thresholds and ratios of shared quantum. These thresholds and ratios determine the task granularity, and they have to be chosen in such a way that granularity must be as coarse as possible to keep all workers working in parallel for as long as possible.

In the example of Fig. 1, W1 has obtained enough work to share and W2 is in the idle state (Fig. 1.a). To share its work, W1 gives its top-most choicepoints (CP1 and CP2) to W2. These choicepoints will become dead for W1 and alive for W2, which backtracks and starts the traversing of the obtained paths (Fig. 1.b). W1 then becomes idle and W2 has created enough work to share. Then, W2 gives its top-most choicepoint (CP1) to W1, which backtracks and takes the next path of its choicepoint (Fig. 1.c).

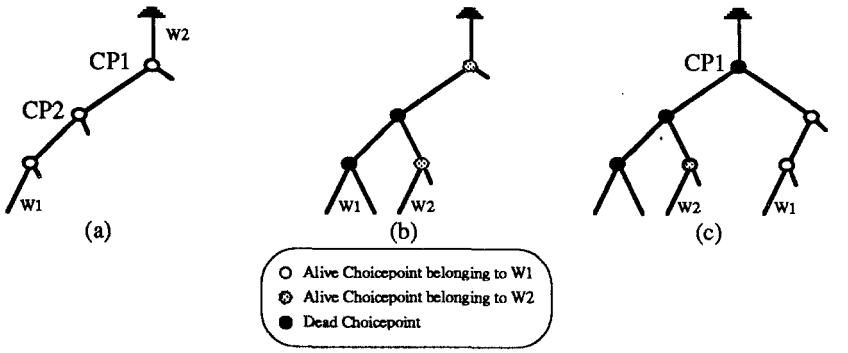


Fig. 1

2.2. Incremental Copy. Worker's State Copying

As it was seen above, this mechanism is a mean to reduce the overhead generated by copying a worker's state to an idle one, so that the idle worker's state mirrors the path to the shared work. It is based on taking into account that the idle worker could already have a part of its state consistent with the worker's state, and therefore, only the inconsistent part up to the work has to be copied. Thus, the choosing of the most suitable worker to share the work seems very important, since the overhead of this operation depends on the size of the inconsistent part. For our explanation of the incremental copy mechanism, it is important to note that during the traversing of a

---

[3] An *alive* choicepoint is such that it has open alternatives to try.

[4] A *dead* choicepoint is such that it doesn't have any alternative to try.

path, the heap and trail of the WAM (it is assumed that the reader is familiar with the memory management mechanism in WAM [7]) will always grow up, but the stack can grow up and down, due to the choicepoint and environment allocation and deallocation.

In shared memory multiprocessor machines a worker can access to a fellow worker's state, thus, a worker knows what part of its state is consistent with that of its fellow worker in an easy and efficient operation. However, in distributed memory systems this operation is a bit more complex and less efficient. When a worker follows a determinate alternative, it records which alternative it has chosen, keeping a trace of the followed alternatives to get to a determinate point in the search tree. This trace, in case of idle worker, holds its absolute position in the search tree, that is, it mirrors the idle worker's state to get to its position. In case of shareable work, the trace holds its absolute position in the search tree, thus mirroring the state to get to the work. This could be enough to know the shared part of its fellow workers' paths, but due to the deallocating of choicepoints, information about the corresponding memory subareas is lost. To solve this lack, when a choicepoint is allocated, and therefore an alternative is recorded, the tops of the heap, trail and stack are also recorded. Thus, each worker keeps a trace of the followed alternatives and their correspondent memory subareas. The *path record* holds this information. As was said above, the allocating of environments can, in some cases, overlap areas in the stack. In such cases, the current top of the stack in the path record will be modified. The incremental copy mechanism uses this information to know which part of the worker's state has to be copied. The trace of the followed path is used by the distributed scheduler to choose the most suitable worker with which to share the work.

To support the incremental copy, the trail mechanism is modified in two aspects. In the first one, whenever a variable is conditionally bound, the *address and value* are stored in the trail. With this information, when the state is copied, these conditional bindings will be installed. This strategy of recording pair address and value is also used in the SRI model [9]. In the other one, a variable will be conditionally bound if the address of such a variable is lower than the current tops in the path record. Normally this behaviour is the same as in the WAM, but in case of the last alternative of a node, when the choicepoint is deallocated, bindings of variables that are lower than the current tops must be recorded in the trail to allow their later installation.

The incremental copy is achieved as follows: After comparing the idle worker's path with the work path, both workers know which part of the path is different. The idle worker has to untrail up to the common part of the paths. Then the memory subareas which correspond with the untraversed path to the work are copied. The trail area is made up of two parts, one to untrail the bindings achieved by the worker during the execution of goals beyond the work, and the other one is used to install those bindings as explained.

Once these areas have been copied and conditional bindings installed, the worker that owned the shared work will discard it. The worker that has received the work will backtrack to the next alternative and will start to traverse the collected set of paths.

An example is shown in the Fig. 2. The trace of traversed alternatives followed by the worker are held in the *Path*, where a '+' and '-' signs mean, from the worker point of view, a dead node and an alive node respectively. The tops of memory for each node are stored in *MemTop*. This is necessary due to the deallocation of choicepoints, as it was said above. The top of the stack for each node is shown with an 'S' and a number to illustrate the depth of the node. The tops of the heap and trail

follow the same terminology with an 'H' and 'T' respectively. Let's suppose that W2 wants to share part of its work (its fourth node) and W1 is the selected worker with which to do it. The comparison of both workers' *Paths* shows that both have traversed the same path up to the second node, where they followed different alternatives. Therefore the workers' state from the beginning up to the second node is almost consistent (the inconsistent parts will be restored during the installation of trailed variables). The idle worker (W1) will backtrack up to the common part (untrail up to T1), and then, the related memory subareas from W2, pointed from *MemTop*, will be copied to W1 from the second node (S1'H1'T1') up to the choicepoint with available work (S3'H3'T3'). The trail from T3' to the top will be copied too, in order to uninstall the bindings achieved by W2 during the traversing of paths beyond the work. The given choicepoint will become dead from the W2 point of view. Then W1 will install the bindings of variables stored from T3' to T1' and it will backtrack to the next alternative. To simplify this example, we have supposed that the stack is not overlapped during the traversing of paths.

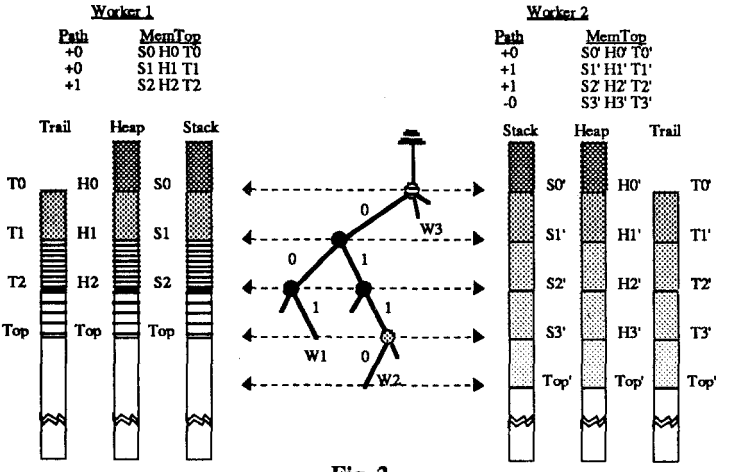


Fig. 2

2.3. Distributed Scheduler

This section will show how workers can all work together and synchronize with one another to share the created work between all workers in the system.

On distributed systems that are not fully connected (which is our case), the communication between workers that are not adjacent is much less efficient than between adjacent ones. Therefore, a strategy that favours cooperation between adjacent workers is necessary. To achieve this, in our model a worker is surrounded by levels of neighbourhood, where each level will hold workers that are at the same distance from it (Fig. 3).

The distributed scheduler has several priorities when the work is going to be shared. First, the sharing of the work will be achieved with available workers belonging to the closest level. Second, in this level, the worker which is closest, within the search tree, to where the shareable work has arisen will be selected.

Then, for workers belonging to the same level of neighbourhood, it will be selected such whose position in the search tree is the nearest to the position in the search tree where the shareable work have arisen.

For each level of neighbourhood, there will be a suitable work threshold and ratio of sharing. These thresholds and ratios have to be chosen taking into account the communication costs for such distances.

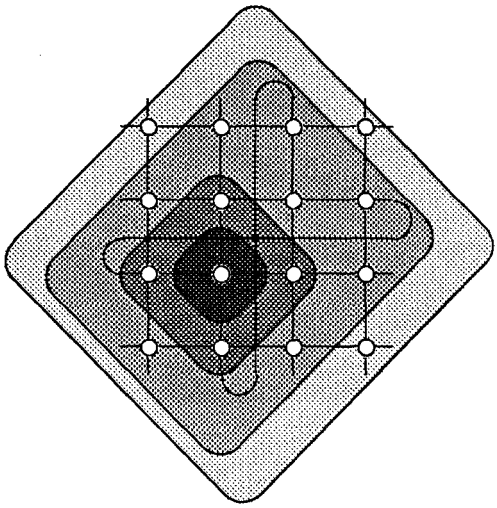


Fig. 3

The work sharing protocol could be roughly seen as follows: When a worker has some shareable work, it will select the most suitable idle worker with which to share that work. The idle worker will choose the most suitable work from the proposals received, and will reject the others. If a work is rejected, the worker that owns that work will select another idle one and the process will be continued as above.

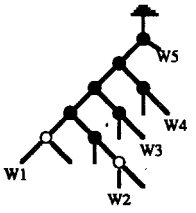
To correctly understand the following explanation, some terminology must be made clear.

- *The path of an idle worker* contains the alternatives followed in order to reach the position in the search tree where it became idle.
- *The path to a work* contains the alternatives followed up to the position in the search tree where the work was created.
- *The most suitable idle worker* for a determinate distance with which to share work is such whose path has more common branches (nodes) with the work one than any other idle worker in that distance.
- *The most suitable work* within a given distance from the point of view of an idle worker is that whose path from the common part up to the work has the least amount of branches (nodes).
- *Work becomes shareable* for a determinate distance when the number of alive choicepoints is higher than the specified threshold for that distance.

Below is shown an overview of the main steps that workers follow to carry out the scheduling of work.

- When a worker runs out of work, it sends its state (Idle) and its path to distance-1 neighbour workers, and *only* its state to the other ones.
- When a worker has created some shareable work to share with distance-1 neighbour idle-workers, it will send the work path to the most suitable distance-1 idle-worker.

- If there are no distance-1 idle-workers and there is shareable work for distance-2, it will request the unknown paths of distance-2 idle-workers. If there are no distance-2 idle-workers, the process will go on with the next level and so on.
- If meanwhile some distance-1 workers have become idle, the process will continue with these ones, if not, the process will proceed with the next distance idle workers as it was seen above with distance-1 ones.
- When an idle worker receives some work path messages, it will select the most suitable work from the nearest level of neighbours, and it will request it. The other received work paths will be rejected.
- On the other hand, when a worker that has sent a work path message to an idle one, receives a reject message, it will continue looking for an idle-worker with which to share the work.
- The worker that receives the work request message will share its work with that idle worker as seen above.
- When an idle-worker gets a piece of work, it will send its new state (Working) to the others workers.
- Computation will finish when the master worker detects that all workers are in the idle state.



(a)

W1	W2	W3	W4
+0	+0	+0	+0
+0	+0	+0	+1
+0	+0	+1	+1
+0	+1	+1	
-0	+1		
	-0		

(b)

Fig. 4

In the example of Fig. 4, let's suppose that W3 and W4 are at the same distance (in the same level of neighbourhood) from W1 and from W2. Then W1 and W2 will select W3 to share their works, since it is the worker that has more common branches (nodes) with the available work from W1 and W2. However W3 will select W1 and will reject to W2, then W2 will select W4, and this one will request that work to W2.

2.4. WAM Extensions

The extensions to the WAM could be divided in three main groups: worker path operations, trail mechanism modifications and distributed scheduler support. The basic structures in WAM have been kept unmodified.

Worker path operations modify four WAM instructions, which keep their original definitions with some additions to record the traversed path. These modified instructions are *Try*, *Retry*, *Trust*, and *Allocate*. Following we will see these additions.

- *Try*: It pushes an open alternative pointing the first branch on top of the traversed path. It also pushes the tops of stack, heap and trail on top of path.
- *ReTry*: It removes the dead nodes from the top of path and increments the pointed branch in the alive node.
- *Trust*: It removes the dead nodes from the top of path, increments the pointed branch in the alive node, and marks such a node as dead.

- *Allocate*: If the current environment address is lower than the top of stack recorded in the top of the traversed path, this top of stack is set up with such lower address.

Trail modifications consist of two aspects. Firstly is whenever a variable binding needs to be trailed, the address and value pair will be stored in the trail. Secondly the limits that specify when the binding of a variable has to be trailed are modified. In our model, the binding of a variable has to be trailed when the address of the variable is lower than the top of the stack or the heap, depending on if the variable belongs to the stack or to the heap. These limits normally coincide with those in the WAM, and they don't coincide in some cases during the traversing of the last alternative of a choicepoint.

The distributed scheduler support consists in:

- Polling and processing of arrived messages.
- Polling and managing of available work for sharing.
- Incrementing the work count whenever a choicepoint is created.
- Decrementing the work count whenever a choicepoint is discarded.

These extensions are supported by some additional structures which are not mixed with the WAM ones. Processing of arrived messages and managing of shareable work follow the cooperation protocol explained in section 2.3.

3. Performance Results and Discussion

This section shows some results after running our system with a set of benchmarks on a distributed memory multiprocessor machine (Parsys Supernode) with 16 Transputers connected to one another following a mesh topology (Fig. 5). Each processor has a Prolog machine with its own local memory, and a small router (Fig. 6). The obtained results are compared with two of the most known multisequential Prolog systems, Aurora [6] and Muse [1] running on a shared memory multiprocessor machine (Sequent Symmetry). In the comparison, it is important to take into account the different stage of development of the systems (Aurora supports sequential side-effect predicates, Muse supports garbage collection, and our implementation is a prototype) and the different architeture on which they are implemented.

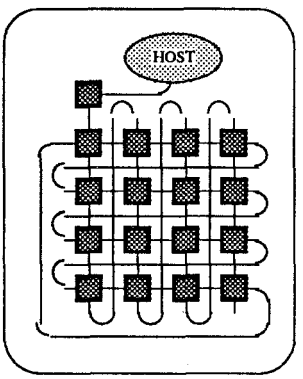


Fig. 5

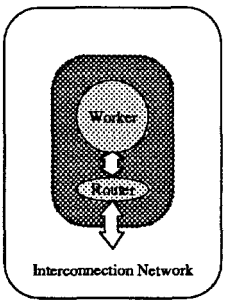


Fig. 6

The set of benchmarks consists in the following programs divided in three groups: the first group is made up of coarse grain parallelism programs.*8-queens-1* and *8-queens-2* are two well known programs from ECRC. *9-queens-b* is a naive generate&test n-queens program attributed to M. Bruynooghe [11], *9-queens-p* is a fused generate&test n-queens program attributed to L. Pereira [11]. The second group is made up of medium grain parallelism programs. *house* is the "who owns the zebra" puzzle from ECRC. The last group is made up of programs that exploit small grain parallelism. *farmer* is the "farmer, wolf, goat, cabbage" puzzle from ECRC. Table 1 shows the run times (in milliseconds) for these benchmarks. All of them look for all solutions of the problem. The speed-up relative to the 1 worker implementation is given in parentheses. The last column shows the run times for the sequential WAM emulator on which our implementation is based, and its speed-up relative to the 1 worker one. The last row shows the sum of the run times for the following programs *(8-queens-1, 8-queens-2, house*20 and farmer*100).*

These results are very encouraging. Our system presents almost linear speed-up for programs that exploit coarse grain parallelism, not very bad speed-up for programs with medium and small grain parallelism. The overhead introduced to support the multisequential model is very low, around 6%.

	1 Worker	4 Workers	8 Workers	16 Workers	Seq. WAM
9-queens-b	2252240.0	569987.0 (3.95)	287953.0 (7.82)	146255.0 (15.39)	2113510 .0(1.06)
9-queens-p	66886.3	16830.6 (3.97)	8643.2 (7.73)	4673.3 (14.31)	63920.6 (1.04)
8-queens-1	17378.5	4424.1 (3.92)	2404.6 (7.22)	1489.3 (11.66)	16686.2 (1.04)
8-queens-2	42651.6	10753.3 (3.96)	5793.5 (7.36)	3400.7 (12.54)	41419.3 (1.02)
house*20	13391.9	4155.5 (3.22)	2679.3 (4.99)	2026.1 (6.60)	11094.5 (1.20)
farmer*100	7187.7	2822.7 (2.54)	2675.7 (2.68)	2886.9 (2.48)	6437.3 (1.11)
Σ	80609.7	22155.6 (3.63)	13553.1 (5.94)	9803.0 (8.22)	75637.3 (1.06)

Table 1.

	1 Worker	4 Workers	8 Workers	15 Workers	Sicstus 0.6
9-queens-b	850000	217400 (3.90)	113000 (7.52)	59700 (14.23)	-------
9-queens-p	24400	6800 (3.58)	2700 (4.60)	2700 (9.03)	-------
8-queens-1	7831	2000 (3.92)	1010 (7.75)	559 (14.01)	6770 (1.16)
8-queens-2	20700	5179 (4.00)	2600 (7.96)	1411 (14.67)	16450 (1.26)
house*20	5021	1480 (3.39)	940 (5.34)	769 (6.53)	4220 (1.19)
farmer*100	3620	2110 (1.72)	2110 (1.72)	2390 (1.51)	3060 (1.18)
Σ	37173	10769 (3.45)	6660 (5.58)	5129 (7.24)	30500 (1.21)

Table 2. Run times for Aurora on Sequent Symmetry [1,11].

	1 Worker	4 Workers	8 Workers	15 Workers	Sicstus 0.6
8-queens-1	6910	1740 (3.97)	880 (7.85)	490 (14.10)	6770 (1.02)
8-queens-2	17540	4419 (3.97)	2240 (7.83)	1209 (14.51)	16450 (1.07)
house*20	4390	1331 (3.30)	840 (5.23)	689 (6.37)	4220 (1.04)
farmer*100	3199	1399 (2.29)	1429 (2.24)	1429 (2.24)	3060 (1.05)
Σ	32039	8889 (3.60)	5389 (5.94)	3817 (8.39)	30500 (1.05)

Table 3. Run times for Muse on Sequent Symmetry [1].

In the comparison with Muse, the overhead introduced to support the parallel model is very similar. The speed-up is a bit lower in our system. This could be due to the fact that the scheduler in a distributed environment is less efficient than the Muse scheduler, which uses shared memory. The speed-up factor of our system is better for small grain parallelism programs than in Muse or Aurora.

With respect to the comparison with Aurora, our system has lower overhead, and the speed-up for the 9-queens programs is better in our implementation. For the 8-queens program, the speed-up of Aurora is better.

The results of these comparisons must be considered only as a point of reference about the proposed model.

5. Conclusion and Future Work

In this paper, we have presented an efficient OR-Parallel Prolog model for distributed memory multiprocessor systems. This model has been implemented on a 16 transputer network with very good performance results. A simple WAM emulator has been adapted to implement our model in an easy way. The overhead introduced to support the multisequential model is very low, around 6%, and the speed-up is close to linearity for coarse grain parallelism problems. However, at this stage of development, it doesn't support side-effect predicates.

Our model has been compared with two of the best and most known OR-Parallel Prolog systems. However, these Prolog systems run on shared memory multiprocessors and ours runs on distributed memory multiprocessors. Due to the different architecture on which they are implemented, and to the different stage of development, comparison results must be considered only as a point of reference. However they are very encouraging.

Our next steps could be the improvement of the general characteristics of the model, the supporting of sequential side-effects, and the development of a model to enhance our multisequential model with exploitation of Stream AND-parallelism on distributed systems.

References

[1] Ali, K. Karlsson, R. "The Muse Or-Parallel Prolog Model and its Performance". NACLP 90. pp 757-776.

[2] Baron, U. Ratcliffe, M. Syre, J. "The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results". Proc. Int. Conf. on Fith Gen. Comp. Sys. 1988. ICOT 1988.

[3] Ciepielewski, A. Haridi, S. "A formal model for OR parallel execution of logic programs". IFIP 1983.

[4] Conery, J.S. "AND Paralelism and Nondeterminism in Logic Programs". New Generation Computing. 1985.

[5] Kacsuk, P. "A Parallel Prolog Abstract Machine and its Multi-Transputer Implementation". The Computer Journal, Vol 34, No. 1. 1991.

[6] Lusk, E. Warren, D. Haridi, S. "The Aurora Or-Parallel Prolog System". University of Bristol. TR-90-07.

[7] Warren, D.H.D. "An Abstract Prolog Instruction Set". Technical Note 309, SRI International, 1983.

[8] Warren, D.H.D. "Or-Parallel Execution Models of Prolog", TAPSOFT '87, Springer Verlag, LNCS 250.

[9] Warren, D.H.D. "The SRI-model for Or-Parallel Execution of Prolog - Abstract Design and Implementation Issues". 1987 IEEE Int. Symp. in Logic Prog., San Francisco.

[10] Zhang, K. Thomas, R. "DIALOG - A dataflow model for parallel execution of logic programs". Future Generation Computer Systems. North Holland 1991.

[11] Tick, E. "Parallel Logic Programming". The MIT Press.1991.