Journal of Programming Languages 2 (1994) 109–124

Implementation and evaluation of OR parallel Prolog models on distributed systems

V. BENJUMEA, M. ROLDÁN and J. M. TROYA

Dpto Lenguajes y Ciencias de la Computación, Universidad de Málaga, Plaza el Ejido s/n. 29013 Málaga, Spain

Received 1 March 1993

A Prolog parallel interpreter with which to evaluate several approaches in distributed OR parallel implementations is shown. Two execution models based on a process tree, which differ mainly in the process granularity, are considered. Furthermore, two memory management strategies based on a closed environment are examined. The evaluation method consists of comparing real results obtained after running four different versions which combine the different issues.

The interpreter has been implemented on a 16 transputers Parsys Supernode, using Occam as the development language. The process and memory management have been implemented in a dynamic way, allowing improvement of the system's efficiency and reduction in memory waste. A dynamic load balancing mechanism has been implemented. Some statistics made using the interpreter show a speed-up of 3.75 for 4 processors and 12.57 for 16 processors.

Keywords: Parallel interpreter, logic programming, AND parallelism, OR parallelism, distributed memory systems, closed environment memory model

1. Introduction

Logic programming was a crucial advance in computer science, allowing programmers to focus on specifying the relationships between problem constituents, and raising the programming abstraction level. However, its practical applications were conditioned to obtain efficient languages and implementations. This was achieved with the development of the Prolog language and efficient sequential implementation techniques [1]. This can be improved using the advantages of the new parallel architecture. This approach is highly appropriate, since logic programs have a significant amount of implicit parallelism. This can be exploited transparently as it is unnecessary to declare it explicitly.

There are two main sources of parallelism in Prolog: AND parallelism (which executes in parallel a conjunction of goals) and OR parallelism (which explores in parallel the different branches in the Prolog search tree).

AND parallelism has been exploited in languages keeping the Prolog syntax and semantic

0963-9306 © *1994 Chapman & Hall*

[2–4], and logic languages based on communication between concurrent processes, such as Parlog [5], GHC [6] and Concurrent Prolog [7].

OR parallelism has been widely studied since the first works achieved by Ciepielewski and Haridi [8] and others. There are several approaches, including the process tree model [3] (usually used with AND parallelism), multisequential models such as Aurora Prolog [9] and Muse [10], and data flow models [11].

The main studies on exploiting parallelism in logic programs have been based on shared memory systems [9, 12]. However, distributed systems are becoming more and more important, while logic programming is an almost unexplored field in this kind of architecture. The main issues to be considered in order to correctly exploit some kind of parallelism are the execution and memory management models, and their suitability to the underlying architecture.

This paper deals with the evaluation of two execution models based on the process model. One model is based on the classic OR tree. The other model uses the OR tree combined with all-solutions AND parallelism [13, 14]. However, the last model is just a way to exploit small-grain OR parallelism. In this model, processes are linked, constituting a pipeline of solutions. Another goal is the evaluation of memory management strategies which will be affected by two aspects, the execution model (based on processes) and the distributed architecture, in which remote memory accesses are very expensive. Therefore it is convenient that the information a process needs is stored in the processor where the process is running. Because of this, we have studied two strategies, the closed environment model [15] and a variation of this based on the use of local memory to the processes.

It is proposed that these goals are achieved by following a practical approach, that is, the implementation of different versions in which only the aspect to be evaluated is changed, and to compare directly the real results obtained from the same Prolog programs. To achieve this goal in a simple and reliable way, the implementation has been developed using a layer architecture, so that each version just differs in the layer affected by the aspect to evaluate, ensuring that the part of the system which has not changed always has the same influence on the run times.

The distributed system was built using the transputer as the basic processing unit, since it is specially adapted to the desired characteristics. Occam was used as the programming language since it was specially designed to exploit transputer characteristics. Transputer meshes with 1, 4, 8 and 16 processors have been implemented on a Parsys Supernode. To use the processor resources in an efficient way, dynamic memory and process management were emulated. Other implementations with similar characteristics use the resources in a static way, with an excessive waste of resources, restricting the system's capacity.

Because communication between processors is a significant source of delay, an important design goal was to reduce the passing of messages through the interconnection network. To have suitable load balancing, a dynamic load distribution mechanism based on process creation was used, since process migration is very expensive in distributed systems.

The rest of the paper is organized as follows: Section 2 describes system architecture characteristics, and its different layers; Section 3 presents the dynamic load balancing mechanism used; Sections 4 and 5 describe respectively the execution and memory management models used; Section 6 shows some results among the different versions, and draws comparisons with other similar implementations. Finally, Section 7 presents the conclusions.

2. System architecture

The interpreter has been developed for a system made up of one processor (master transputer) running as a user interface and N processors (transputers) whose main goal is to look for solutions to a specified query.

The system can be in two different phases: analysis and resolution. In the analysis phase, the source code for the Prolog program is checked and internal code is generated for its later interpretation. The internal code is broadcast to each processor in the network. This allows dynamic load balancing, whereas a static distribution of code is unsuitable for this. The analysis task is carried out by the master transputer. The remaining processors broadcast the internal code through the network. During the resolution phase, the network processors traverse the search tree, seeking solutions for the specified query. The master transputer reports to the user about the solutions found.

The resolution system is made up of a processor network, where each processor will have the same internal configuration (Fig. 1). It consists of two main modules, the resolution module, whose goal is to solve the problem in each processor, and the network interface, which deals with the communication between processors, isolating the resolution module from the underlying architecture characteristics (communication and network topology). Occam processes are used to implement the network interface and the resolution module. However, to achieve flexible process management, the processes which make up the execution model during the resolution of a Prolog program are not mapped on Occam processes. It is the implemented system which provides primitives for the creation and scheduling of logic processes, placed in the layer relating to this task.

The system is made up of four layers, as shown in Fig. 2. Each layer offers services to the adjacent ones. This organization allows independence between the different functions that manage the processes and communications. The function of each layer is described below.

The aim of the interpretation layer is to execute the process that achieves the problem resolution. It is a state machine implementing the chosen execution model (OR parallelism or all-solutions AND/OR parallelism). It uses the services provided by the next layer.

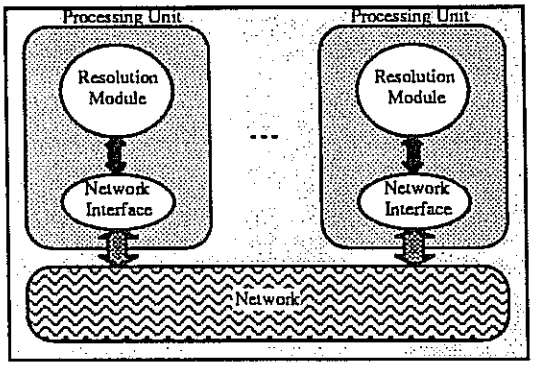

Fig. 1. The resolution system

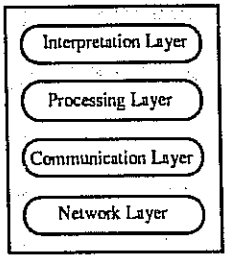

Fig. 2. The system layers

The processing layer is made up of three functional units:

- process control unit
- inter-process communication unit
- local memory management unit.

The process control unit deals with local and remote process creation, and their scheduling. The inter-process communication unit manages the communication between local and remote processes, providing buffered asynchronous communication. In the case of remote processes it will use the services provided by the lower layer. The local memory management unit provides services to access the local memory and manages the memory model that supports the OR parallel execution.

The communication layer handles message management between processors, regardless of the network topology. It centralizes the communications between processes in different processors (remote process activation, goal solution, environment, load balancing, etc.), and creates the messages that will be delivered to the destination processor.

The network layer manages and routes the message passing between processors, hiding the network topology from the upper layers. It knows the path to deliver a message to its destination.

It is important to note that communication between layers is implemented by means of calls to the services provided, except between the communication and the network layers, where it is achieved by message passing, since such layers belong to different Occam processes.

An example of the interaction between the layers can be seen in Fig. 3a. The interpretation layer is executing a process which wants to create new processes. It calls the process creation service provided by the process control unit. This will apply the load balancing algorithm to see where to create the new processes. If these must be created in the local processor, the process control unit asks the memory management unit for the local process creation service, which allocates the new process status in the system memory. If the new processes must be created in a remote processor,

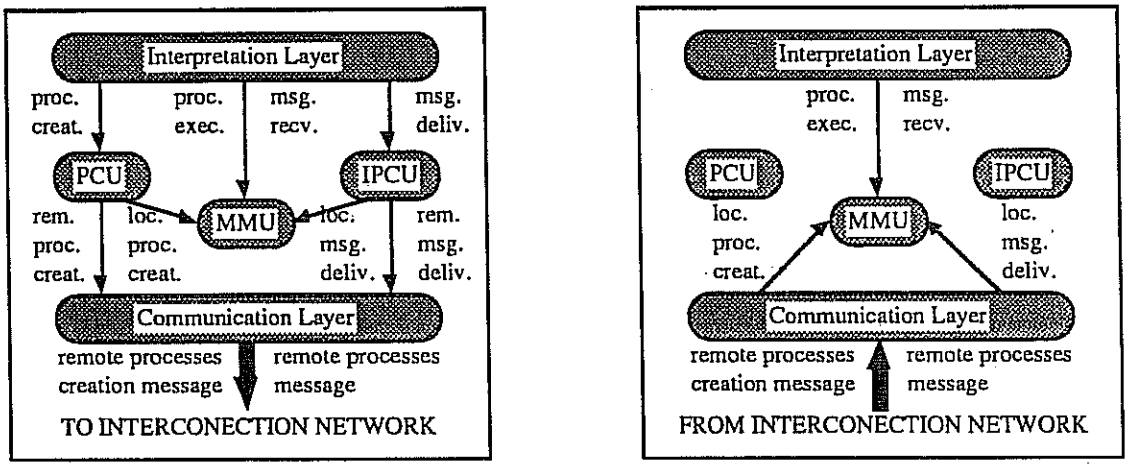

Fig. 3. Examples of interaction between layers

the process control unit asks the communication layer for the remote process creation service, which sends that message to the network. The message is delivered through the network layer and the interconnection network up to the communication layer in the destination processor (Fig. 3b). It is processed and the memory management unit is asked for the local process creation service.

Let us suppose that the process running in the interpretation layer wants to send a message to another process (Fig. 3a). It asks the inter-process communication unit for the message delivering service, which calls the local message delivering service provided by the memory management unit, if the destination process resides in the same processor. The memory management unit allocates the message in the process associated buffer. If the destination process resides in a remote processor, the inter-process communication unit requires the communication layer to send the appropriate message to the network through the remote message delivering service. This message will get to the communication layer in the destination processor (Fig. 3b). It is processed and the memory management unit is asked for the local message delivering service.

3. Load balancing

Load balancing is based on process creation, which is carried out in the most suitable processor according to an evaluation function. When the time required for process migration is a significant proportion of the average execution time of a process, it is not advisable to allow process migration. These models have medium and fine granularity and, therefore, in this system, process migration does not exist, and a process will be executed on the processor where it was created.

When a process needs to create child processes, such a creation is made in a local or remote processor, depending on the estimated cost of each case and, furthermore on the load in each candidate processor. Since remote creation has a high cost in a distributed system, the local creation of processes is favoured.

Creation and communication costs depend on the distance, on the interconnection network, between the processors involved in such a communication. Distant processors obtain unfavourable results from the evaluation function, and are never selected.

Since there is no process migration and adjacent processors are the preferred candidates, when a message is generated it will always be directed to the local processor or to some processor directly accessible on the interconnection network, so that the traffic on the network is not very high, and is carried out with a very low routing cost.

Because the load balancing algorithm is dynamic, the selected processor will depend on the system state at the process creation time. Such a selection is based on two parameters:

> N, the number of active processes N_A plus inactive processes N_I in the candidate processor; and C_{ij}, the estimation of the cost that a process located in processor i has to pay to create a process in processor j.

Each processor must know the parameter N of the candidate processors. To achieve this, it sends information periodically about active and inactive processes to its neighbour processors.

The cost matrix C is a constant of the system that will be estimated depending on how much either the process spread by the network or its local creation is to be favoured.

To get the results shown in section 6, the following evaluation function is used.

$$F(i,j) = ((N_A[j]^*weight) + N_I[j])^*C_{ij} \quad \text{where } i = \textit{local processor}; \textit{ and } j = \textit{destination processor}$$

with parameters

$weight = 2$

$C_{ij} = 1$ (i = j)

2 (i and j are neighbours)

∞ (otherwise)

4. OR parallelism models

Two execution models based on process tree have been implemented. In these, the program is solved using concurrent processes communicating to one another through message passing. Both models differ mainly in the process granularity. The model with larger granularity has less overhead. However, this may be a drawback with regard to the exploited parallelism, as it increases the probability of a processor being idle since there will not be available active processes to be executed. To evaluate that behaviour, execution models based on the classical OR tree, using medium granularity processes, and on the OR/AND all-solutions tree, which reduces the process granularity, have been implemented.

4.1 *OR execution model*

In this model, a process tries to solve a clause body. So, when it is activated, it receives the clause whose body must be solved and its variable bindings (environment). A process tries to solve the first goal, so it creates as many processes as there are unifiable clauses (exploiting OR parallelism) and it waits for the first clause solution. Then it tries to solve the second goal of the solution and so on until the last goal. When it receives the solutions from the last goal, it sends them to its father as clause solutions. When the former obtains the next solution from the previous goal, it will look for further goal solutions, and so on.

This is explained in more detail in the example in Fig. 4, where process P1 tries to solve the initial query by receiving its initial environment. It tries to solve the first subgoal, creating P2 and P3. It gets the solution from P2 and activates P4 (which tries to solve the clause that unifies with the second subgoal). When it receives the solution from P4, it activates P5 and P6. Meanwhile, the other processes go on searching for more solutions. When it receives the solution from P5, this will be the first solution to the initial query. The solution from P6 will be the second one. Since there are no more solutions for the last subgoal, it will get another solution from the previous one (P4).

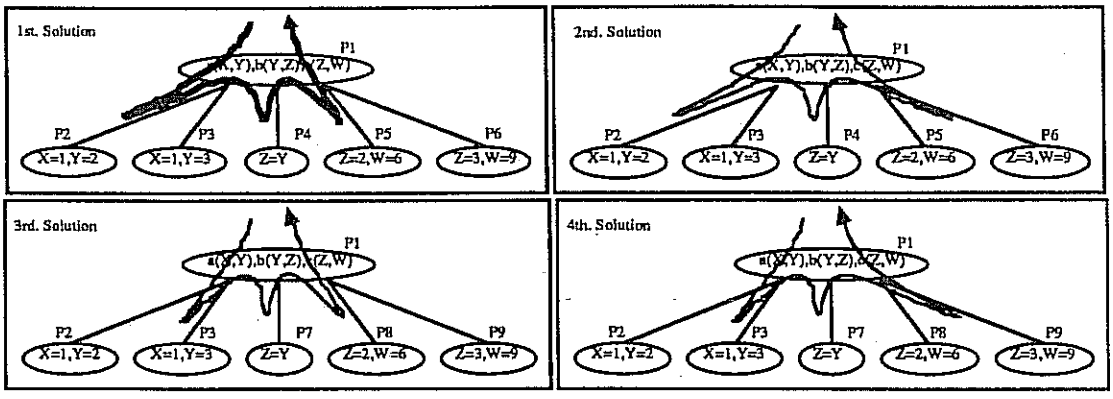

Fig. 4. OR execution model

However this hasn't got any solution, so it has to go back to the next solution for the first subgoal, which comes from P3. With this it will activate P7 (which tries to solve the second subgoal). With its solution it will create P8 and P9, whose solution will be the third and fourth solutions for the query respectively. The solution flow is shown by a thick line.

4.2 *All-solutions AND/OR execution model*

We have started from a basic model, which uses the AND/OR tree (Fig. 5a). In this model AND processes, which deal with the solutions of the body in a clause, can be distinguished from OR processes, which deal with each literal in a clause. The *all-solutions* AND parallelism is exploited by the AND processes, which manage, in parallel, different solution flows. The OR parallelism is exploited by the OR processes, solving different clauses in parallel. Figure 5a shows that direct communication between sibling processes is not possible. It is the father process which deals with such communication.

A very modified version of this has been implemented here, with the following main differences.

- The communication scheme of the basic model is not flexible, since all messages must be managed by AND processes. To improve it, the model proposed allows direct communication between sibling OR processes, so many unnecessary messages are avoided.
- The proposed model only uses one kind of process. AND processes are not used, but its functionality is subsumed by OR processes, so that the process granularity is increased.
- To manage backtracking in the basic model, many processes have to be destroyed, whereas it is probable that they will be needed later, and will therefore frequently have to be recreated. To avoid this, processes are not destroyed but suspended. They are activated when they receive a new environment with which to obtain more solutions. So, the time spent on process management is reduced.

The Fig. 5b shows the execution tree resulting from the proposed model.

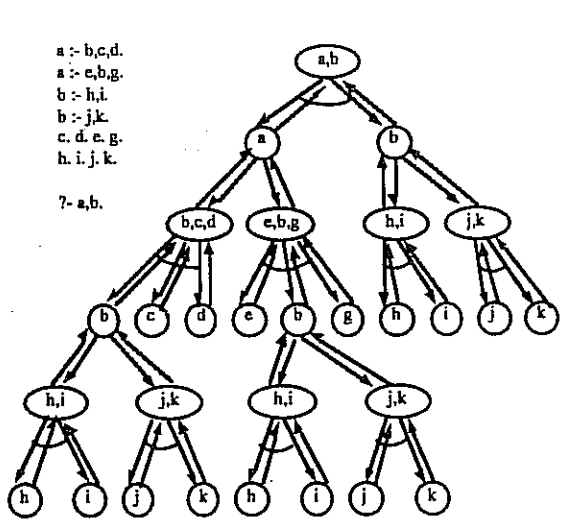

Fig. 5a. Process tree and messages in the original model

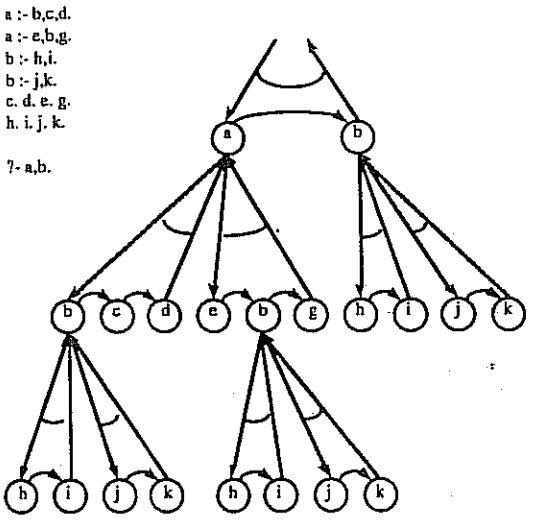

Fig. 5b. Process tree and messages in the proposed model

4.3 *General behaviour*

In the proposed model, a process tries to solve a goal. To achieve this, the process receives the goal environment. For each clause that unifies with the goal to be solved, it creates processes to solve the goals belonging to the clause body, linking them in a pipeline of processes where the goal solutions are environments for the next one. The process solving the first goal receives the environment from its father, and the process that solves the last goal will send the solutions to its father.

An example may be found in Fig. 6, where P1, P2 and P3 processes are created to solve the query subgoals. These wait to get the environment(s) in which they will look for solutions. So, P1 receives the initial query environment and it creates P4 and P5 to solve the body of the c1 clause, and P6 and P7 for c2. The solution obtained from P5 is sent to P2 (after a closing operation) and, with this, it creates P8. The solution coming from P8 is passed to P3, which creates P9, P10, P11 and P12. The solutions from P10 and P12 are the first and second solutions for the query respectively. Because there are no more solutions, it gets the next one from P2, which creates a new process, P20, from the solution sent by P7 to P1 and from this to P2. The solution found by P20 is passed to P3, which creates P21, P22, P23 and P24. The solutions from P22 and P24 are the third and fourth solutions for the query. The solution flow is shown by a thick line.

In both execution models, a process will be created with information regarding the goal or clause to be solved, and then it will wait for the environment from which it will seek solutions. In the latest model, a process will receive several environments from which it will generate suitable solutions. Therefore, just one creation is necessary.

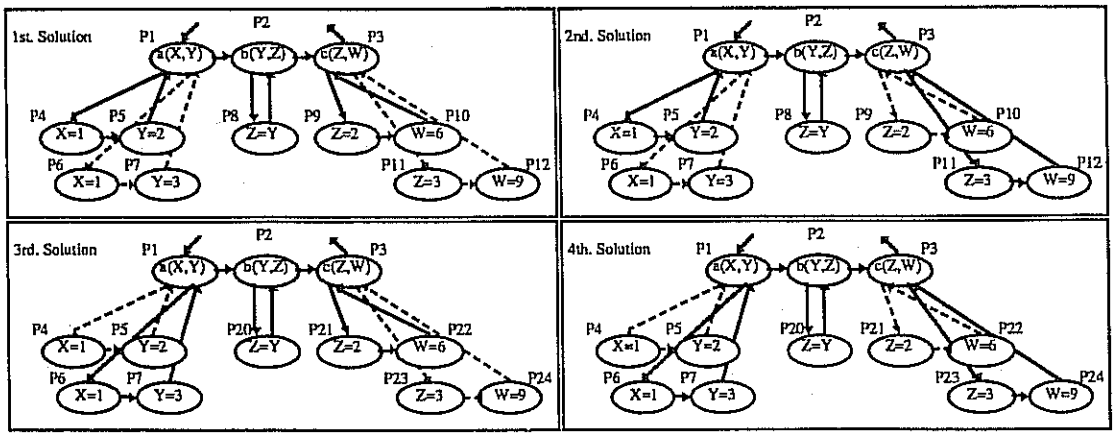

Fig. 6. All-solutions AND/OR execution model

5. Memory management

Implementing the execution model in an efficient way requires a memory management strategy suited to the execution model and to the underlying architecture (shared memory or distributed memory). In distributed systems, it is advisable that each process be able to get the information necessary to solve its goal from the local processor memory, since accesses to remote memories are very expensive. Because of this, we have chosen the closed environment model proposed by Conery [15] and a variant of this as memory models. In these models, all the information necessary to achieve the unification is kept in two environment frames in such a way that these frames are isolated from the rest and are in the processor where the process is running. These models are oriented to an execution model based on processes, since each environment frame keeps the information needed by a process.

Two implementations based on Conery's model have been developed, as follows.

- Local model: each process has its own local stack and heap. These are kept in an adjacent space in the processor memory.
- Global model: it follows the main guidelines described in [15], in which there is a stack and heap per processor shared for all the local processes, so that the structures created in the heap can be shared by several environment frames.

5.1 *Closed environment model*

This model was proposed by Conery [15] to solve variable binding conflict in OR parallelism on non-shared memory architecture using a process-tree execution model. The binding environment seen by a process is restricted to one or two frames. These are organized in such a way that they hold all the information needed for unification.

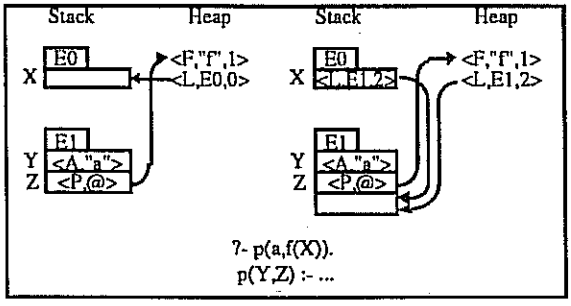

Fig. 7. Example of unification and closing

A closed frame is such that no link originating in it de-references into slots belonging to other frames.

To guarantee that the two frames that will be unified are closed, it is necessary to follow several rules, as follows.

- If two closed frames, one from the goal environment (top frame) and one from the candidate clause (bottom frame), are unified, the environment closing operation can transform them so that one will remain closed with respect to the other one.
- The bottom frame for a candidate clause will initially hold only unbound variables. Therefore it is a closed frame. The frame of the initial goal statement holds unbound variables, so that it is a closed frame.
- After unifying a goal with the head of a unit clause, the top frame must be closed. This frame can be used to solve siblings of the goal or it can be used, if the goal is the latest in the clause, as a solution for the clause.
- After unifying a goal with the head of a non-unit clause, the bottom frame will be closed with respect to the top frame. This bottom frame becomes the top frame for the first goal in the body of the clause.
- After solving the last goal in the body of a clause, the calling goal's frame will be closed with respect to the bottom frame made for the clause.

General characteristics

In each processor, the system memory keeps information about process status and messages. Process status and message heads are fixed size structures (managed as a stack with a hole list) holding pointers to an area where variable size structures are stored (managed as a stack with holes). Every local process can have access to these areas, so if a process wants to send a message to another process in the same processor, it is an operation in the local memory, which is more efficient than message passing between Occam processes.

In addition to those structures, both models use several areas to hold the data generated during the Prolog program execution. In the local model the stack and heap of each process are stored in the area of variable size structures. However, in the global model, together with

the above areas, there is a global memory area, shared by all the processes allocated in the same processor, holding their stack and heap.

Global model implementation

The closed environment model, as proposed by Conery, is implemented. System memory is composed of three areas: stack, heap and process control.

- The stack holds frames generated during program execution. Each frame holds a frame identifier, and a slot for each variable of the clause that it represents, and for those that are newly created during execution time.
- The heap holds dynamic structures generated during execution time. All the processes allocated in the same processor may access to this area. Therefore, structures may be shared by several frames of the stack.
- The process control area holds the auxiliary and process control structures.

When a process receives a frame, it will make as many copies of it as there are clauses that unify with the literal corresponding to such a frame. It requires copying of frame variables and of the structures pointed from that frame.

The unification algorithm will work with the copied frame and the candidate frame, which initially will have unbound variables, Since the heap is shared among the processes allocated in a processor, to unify a variable with a structure does not require a copy operation. However it does require an operation to bind the variable to the structure in the heap.

De-referencing and frame management

The memory management requires frequent frame duplications and consequently, duplication of related structures. This operation must be carried out guaranteeing a later unambiguous de-referencing. To differentiate the original frame from the copies, a different frame identifier could be used in each frame. However, that implies traversing the referenced structures to change references to the original frame for references to the copied frame, which is an inefficient operation.

The proposed de-referencing mechanism uses the same frame identifier in the original and copied frames. It is only necessary that the top and bottom frames have a different identifier. We use two tables that always allow the access to the requested frame address.

Ref_top is a table with as many slots as the body of a clause has literals. Each slot contains a pointer to the top table that allows access to all the frames related to the literal.

Top is a table with as many slots as there are clauses that unify with a literal. Each slot contains the address of the implied frame in such unification.

The de-referencing mechanism is very fast since during unification the absolute addresses of implied frames are known.

Local model implementation

The local stack of processes is assigned as a structure, which holds a slot for each variable in the clause. The heap, stored adjacent to the stack, will hold dynamic structures. The stack and heap

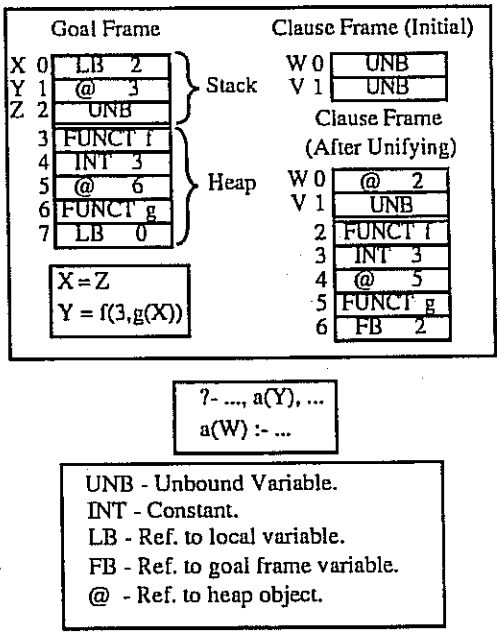

Fig. 8. Example of unification

reside in the processor system memory, which is accessible to local processes. This allows the local memory to be shared, so the size of the local stack and heap is not static, but it is allocated in a dynamic way according to needs.

The process receives the clause environment in which it will try to solve its subgoal. It will try to unify a copy of the clause environment with the new environment created for the candidate clause. During the unification, dynamic structures involved in it will be copied from one environment to the other. The references between variables will always be from the new environment to the old one. If the unification succeeds, the newly created environment will hold the bindings for the unified clause (references to variables in the older environment are treated as unbound variables) and it will be sent to the first subgoal process for the clause. Solutions for the clause are received from the last subgoal process. The bindings for the variables that referenced unbound variables in the older environment are copied to this one. This environment will now be the environment solution for the subgoal that tries to solve the process, and it will be sent to the siblings if there are more subgoals in the clause or, if this is not the case, it will be sent to the father as a clause solution.

To illustrate the unification mechanism in this model, we can see in Fig. 8 an example of unifying a(Y) with a(W). The goal environment will be a copy of the clause environment whose body the process is trying to solve. An environment will be created for the clause that is being unified. Initially it will hold slots for the variables, which are unbound, and the heap will be empty.

As shown in Fig. 8, Y refers to a heap structure, and W is unbound. Since each environment has its own local heap, it is not possible to assign a reference to the structure in the goal environment to

Table 1. Results for four system versions

Number of transputers	*Result type*	*OMGM system (V1)*	*OMLM system (V2)*	*OPMGM system (V3)*	*OPMLM system (V4)*
1	Time (ms)	188.69	152.56	218.03	179.13
	Speed-up ratio	1	1	1	1
4	Time (ms)	85.08	69.96	72.59	47.68
	Speed-up ratio	2.21	2.18	3.00	3.75
8	Time (ms)	41.53	34.74	41.18	25.68
	Speed-up ratio	4.54	4.34	5.29	6.97
16	Time (ms)	24.22	18.73	23.65	14.25
	Speed-up ratio	7.79	8.14	9.21	12.57

W, but the structure will be copied in the unified clause environment heap, and W will reference to this. After copying, the references to local variables (LB) in the copied structures will be replaced for references to variables in the source environment (FB).

After unifying, the unified clause environment must be a closed environment, therefore a closing procedure that guarantees that all references are local is necessary. This closed environment, which will hold all the information needed, can be sent to a child or sibling process. The closing procedure is achieved during the unification.

In this model, garbage collection is not necessary since used local environments are de-allocated automatically.

6. Results

Two execution models and two memory management strategies have been combined, resulting in four different versions of the system:

- OR execution model and global memory management (OMGM)
- OR/AND pipeline execution model and global memory management (OPMGM)
- OR execution model and local memory management (OMLM)
- OR/AND pipeline execution model and local memory management (OPMLM)

Layered architecture allows the comparison of different models in a reliable way since each version has only modified the appropriate layer.

Table 1 shows some results of running a Prolog program to solve the 7 Queens problem on networks with 1, 4, 8 and 16 processors, for the four system versions.

To evaluate a parallel execution model, the overhead that is introduced in comparison with a sequential model, and the speed-up in the multiprocessor implementation must be taken into account. Since Tv2 < Tv4 and Tv1 < Tv3, we can say that the execution model based on the OR tree has less overhead. However, the speed-up results are the opposite. Tv4 < Tv2 and Tv3 < Tv1, hence the OR/AND all-solutions model seems more suitable for this kind of architecture.

Table 2. Execution time comparisons

Number of processors	*Program being executed*	*Execution time (ms)*		
		OPMLM V4	*DIALOG*	*CRAI*
1 Transp	database	286.2		780.6
	append1	85	762	
	append2	620	1290	
	qsort	5304	54805	
	na_rev	3318	46612	
4 Transp	database	292.6		234.1
	append1	89	248	
	append2	617	619	
	qsort	5330	20930	
	na_rev	3354	13333	
16 Transp	database	292.2		208.9
	append1	90	194	
	append2	618	617	
	qsort	5385	16680	
	na_rev	3439	7137	

Memory management models may be compared by running different versions for the same execution model on one processor implementation. We have observed that Tv4 < Tv3 and Tv2 < Tv1. In both cases, the time used by the local memory model is 20% less than the global memory model. That is because the structure duplication cost is lower in the local model, where data structures are adjacent in the heap, and such duplications are frequently used to keep the independence between the different search paths.

6.1 *OPMLM vs. other implementations*

Most work on parallel Prolog implementations has been carried out on shared memory architecture. In recent years, studies on distributed implementations have increased. It is not known how many works gave numerical results, however Table 2 shows the execution times compared with implementations in CRAI [16] and DIALOG [17].

CRAI describes an interpreter using an OR parallel model on a network of transputers. Those characteristics are similar to the system proposed here, but implemented in a static way. DIALOG is an implementation of a dataflow model simulation on transputers.

Used programs are not suitable to compare the implementations, since they do not have enough work, and so, the computation time doesn't justify the parallel execution. However, studying Table 2 it can be seen that the OPMLM monoprocessor version is better than the others, but on multiprocessor versions, the proposed system does not offer a speed improvement. This is because the time spent on creating a remote process is time consuming with respect to the global execution time, whereas a static implementation does not waste time in remote process creation. On real

programs, the execution time is longer, and remote process creation time is not significant, whereas a static focus has an excessive memory waste. Results of comparing monoprocessor versions prove that the proposed system has less overhead, and its speed-up for more suitable programs is satisfactory.

7. Conclusions

A parallel Prolog implementation prototype to exploit OR parallelism on distributed systems has been developed. The underlying system is a Parsys Supernode with 16 transputers.

The closed environment model is a suitable memory management strategy on distributed systems. A new version using local memory in the processes has been implemented. It is faster (about 20%) for both execution models (the OR tree model and the OR model combined with all-solutions AND).

Both execution models have been studied to compare their suitability for parallel execution. The results indicate that the former has less overhead, however the latter is a better choice for parallel execution, since it has better speed-up.

The implementation is dynamic, hence it has more flexibility and its utilization of resources is better than models with a static point of view. Layer architecture allow results to be obtained in an easy and reliable way.

Acknowledgements

This work was supported by project CICYT TIC 340/90.

References

1. D. H. D. Warren. *An Abstract Prolog Instruction Set*, Technical Note 309, SRI International, 1983.
2. D. DeGroot. *Restricted AND-parallelism*, Proceedings of the International Conference on Fifth-Generation Computer Systems, ICOT Tokyo 1984, pp. 471–478.
3. J. S. Conery. AND parallelism and nondeterminism in logic programs, *New Generation Computing*, **3** (1985) 43–70.
4. 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, 1986.
5. K. L. Clark and S. Gregory. *PARLOG: parallel programming in logic*, ACM Transactions on Programming Languages and Systems, 1986.
6. K. Ueda. *Guarded Horn Clauses*, TR-102, ICOT, Tokyo, 1986.
7. E. Shapiro. Concurrent prolog: a progress report, *IEEE Computer* **19**, (1986) 44–58.
8. A. Ciepielewski and S. Haridi. *A formal model for OR parallel execution of logic programs*, IFIP, 1983.
9. E. Luske, D. Warren and S. Haridi. *The aurora OR-parallel Prolog system*, University of Bristol, TR-90-07.
10. K. Ali and R. Karlsson. *The Muse OR-parallel Prolog model and its performance*, NACLP 90, pp. 757–776.

11. P. Kacsuk. A parallel Prolog abstract machine and its multi-transputer implementation, *The Computer Journal*, 34, (1991) 52–63.
12. D. H. D. Warren. *The SRI model for OR-parallel execution of Prolog–abstract design and implementation issues*, Proceedings of the 1987 Symposium on Logic Programming.
13. N. Tamura and Y. Kaneda. *Implementing parallel Prolog on a multi-processor machine*, Proceedings of the IEEE International Symposium on Logic Programming, 1984.
14. V. Benjumea. M. Roldán and J. M. Troya. A Prolog interpreter for OR parallelism evaluation on distributed systems, *Euromicro PDP*, 1993.
15. J. Conery. *Binding environments for parallel logic programs in non-shared memory multiprocessors*, IEEE, 1987.
16. M. Cannataro, G. Spezzano and D. Talia. A parallel logic system on a multicomputer architecture, *Future Generation Computer Systems*, North Holland, 1991.
17. K. Zhang and R. Thomas. DIALOG—A dataflow model for parallel execution of logic programs, *Future Generation Computer Systems*, North Holland, 1991, 373–388.

Further reading

INMOS. *Transputer Reference Manual*, Prentice Hall, 1988.
D. Pountain and D. May. *A Tutorial Introduction to Occam Programming*, BSP, 1988.