Side-effects in PloSys OR-parallel Prolog on distributed memory machines*

É. Morel, J. Briat, J. Chassin de Kergommeaux, C. Geyer
Projet CNRS-INRIA APACHE
LMC-IMAG, B.P. 53, F-38041 Grenoble Cedex 9 France
{Eric.Morel, Jacques.Briat,Jacques.Chassin,Claudio.Geyer}@imag.fr
Tl : +33 76 51 46 63, +33 76 51 45 72, +33 76 51 45 75
Fax : +33 76 51 48 46

The aim of the logic programming system PloSys is to offer most Prolog functionalities on distributed memory parallel computers as well as networks of personal computers. The PloSys computational model exploits OR-parallelism. It is designed to implement cut and side-effects while keeping low the number of messages exchanged between processors. It is a multi-sequential model based on copying. Scheduling is centralised as is the management of cut and side-effects. The scheduler keeps up to date the knowledge of the left-most worker in the scope of each cut. A PloSys prototype was implemented and its first results, on IBM SP-1, are promising.
Keywords: Logic programming, OR-parallelism, Distributed Memory, Side-effects, Multi-sequential.

1. Introduction

Logic programming is one of the best suited paradigms to parallel processing and has given rise to a large number of parallel models of wich several were implemented efficiently [9]. By efficiently it is meant here that efficient parallel logic systems execute some programs more rapidly than the most efficient sequential Prolog systems and that, in the worst case (intrinsically sequential programs), parallel execution is not slower than sequential execution. Most OR-parallel logic programming systems were implemented on shared-memory symmetric multiprocessors [2, 18]. Some prototypes were also implemented on distributed memory systems but they do not include the usual Prolog side-effects [3, 4, 15].

The objective of PloSys is to build a "Prolog-like" OR-parallel logic programming system for distributed memory parallel architectures. "Prolog-like" means that PloSys will implement Prolog cut and side-effects. This will ease the parallelisation of applications since applications will be debugged on a sequential Prolog system before being executed in parallel.

The PloSys computational model is multi-sequential: each worker importing work copies the state of the worker exporting work. Scheduling is centralised. The management of cut and side-effects aims at limiting the number of messages exchanged during the execution. A PloSys prototype is currently being tested on distributed memory parallel computer IBM SP-1.8. Its first results are

*This work is subject to a cooperation agreement with KFKI-MSZKI, sponsored by the Balaton project No. 95021 of the French ministeries of Foreign Affairs and of Education

promising. Another research stream, not described in this paper, is concerned with the evaluation of scheduling strategies [16].

The organisation of this paper is the following. This introduction is followed by a general description of the PloSys system. The management of PloSys side-effects is described in the next section. The PloSys prototype is then outlined before a comparison with other OR-parallel logic programming systems.

2. Overview of the PloSys system

The PloSys computational model is a multi-sequential model tailored to distributed memory parallel architectures. Special care was given to keep low the number of messages exchanged during the parallel execution of logic programs. The reason is that, in distributed memory parallel systems, the latency of communications becomes increasingly important relatively to the speed of processors and it becomes more important to reduce the number of messages than their size. The scheduler must on the one hand allocate work to the workers executing the program and on the other hand manage the Prolog cut and side-effects (see section 3.). In the existing prototype, scheduling is centralised although it will be distributed in the future. Each worker was designed so that the management of the cut and side-effects was kept separated from the inference engine. A PloSys prototype is currently being tested (see section 4.1.).

2.1. Computational model

The PloSys computational model is inspired from OPERA [5], without implementing some optimisations tailored for the Supernode architecture used in OPERA. In PloSys, a Prolog program is executed in parallel by several workers, including each an inference engine as well as an exporter and an importer process. The inference engine is a sequential Prolog inference engine, as efficient as possible and therefore based on an implementation of the Warren Abstract Machine (WAM) [23] which executes compiled Prolog programs. An idle worker obtains work from an active one by developing one or several unexplored choices originating from one or several choice points of this worker. To import work, a worker copies the state (mainly the stacks) of the exporter worker up to the choice point (or to the most recently created one) used to provide work. Copy is only possible when the exporter worker is modifying its choice points, in order to simplify the management of the coherency of the bindings of the exported variables. Similarly, the inference engine of the exporter worker remains idle during the export phases.

When receiving work from an active worker, a previously idle worker processes first unbinds non valid bindings[1] using its trail stack, before simulating a fail to execute the most recent untested alternative of the copied state.

2.2. Architecture and functionalities of the scheduler

The design of the scheduler is modular to ease its future evolutions: it includes a skeleton and several independent units. The first unit is dedicated to the distribution of work. The process associated to the work distribution unit collects periodically the state of the workers and distribute work according to a scheduling policy. As in OPERA [5], idle workers search work from "over-loaded" workers whose load is over some pre-defined threshold[2]. Matching idle and over-loaded workers is performed by the

[1] that is bindings that were done by the exporter worker in the development of another, non exported, alternative of the exported state

[2] The determination of this threshold will be subject of future work

scheduler, which keeps an approximate global state of the system using the load information sent by the workers. An idle worker gets from the scheduler the location of an over-loaded worker. The idle worker then requests work from this over-loaded worker, this work consisting of the unexplored alternatives of one or several choice-points. A work request may fail if the requested worker is not any more in the over-loaded state when the request is received (see figure 1).

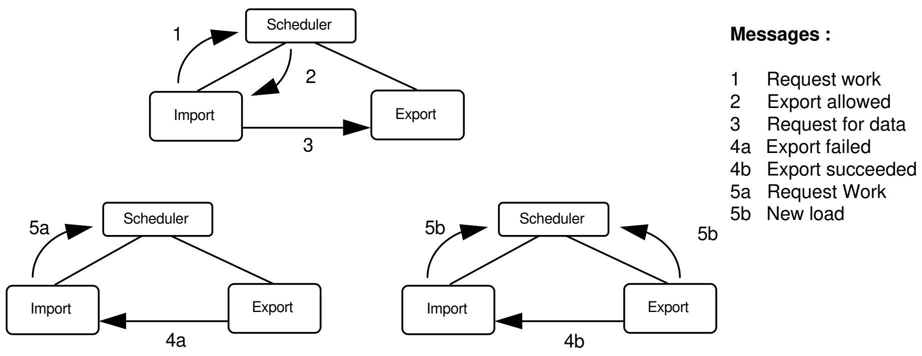

Figure 1: Exchange of work in PloSys

The other units are in charge of coordinating the side effects with each other and load balancing (see section 3.5.). Each unit can be modified independently of the others so that several scheduling strategies can be made available or several side-effect management strategies can be tested or matching of work to workers can be changed (distributed for example).

2.3. Architecture of the PloSys workers

Each worker includes at least three processes : one inference engine, one exporter and one importer. In addition, there exist several processes, specialised for the management of side-effects. Not like in OPERA, the inference engine and the exporter cannot be simultaneously active. Exporting is only possible in some of the states of the inference engine. This limit has a minor consequence on the execution but saves many synchronisations between the exporter and the inference engine processes[3]. Information of the scheduler by the worker is done by regular calls of the inference engine and not by a specialised spy process as in OPERA. As for the scheduler, the implementation of the workers is modular to ease experiments and evolutions (see figure 2).

3. Management of side effects in PloSys

Prolog includes two types of side effects:

- the cut which prunes the branches on the right of the “cut clause” in the AND/OR search tree of the program.
- the other side-effect operators which modify or depend on the environment (I/Os, Prolog data-base, etc.).

[3]Also, DMA not being possible in the SP-1.8 prototype (see section 4.2), exporting and executing the inference engine can not be done at the same time on one worker

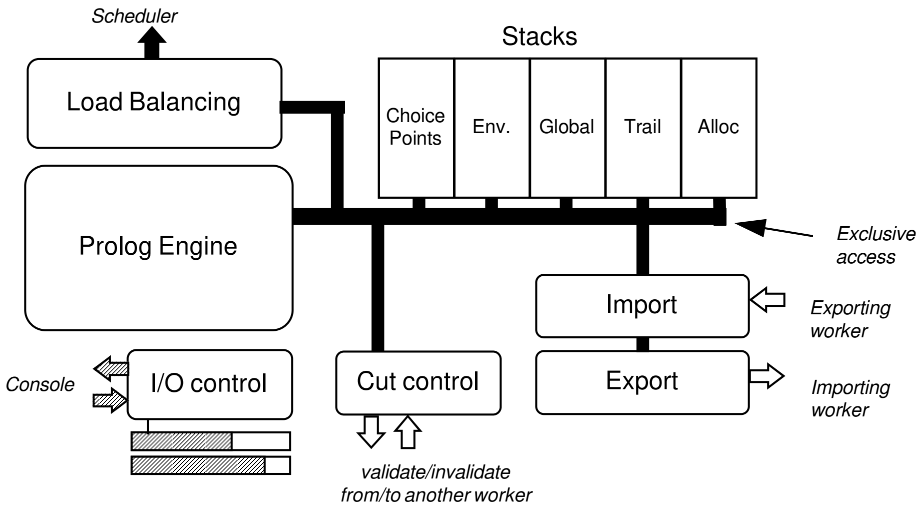

Figure 2: Architecture of the Plosys workers

Management of side-effects There are two possible approaches. Either the semantics of the side-effect operators includes the execution order of these operators. Or, this execution order is determined by the sequential execution order of the Prolog program. The second approach was adopted to keep the programming language untouched. The most simple implementation differs the execution of a side-effect until the processor becomes "left-most" in the execution tree of the program. Another possible solution is to record the operation so that it can be executed when it is correct and meanwhile either to continue the computation (output to a file for example) or to suspend (input operation).

Management of the cut One possible solution is to process sequentially all the branches originating from an OR-node, under the scope of a cut. To avoid the inhibition of parallelism caused by this solution, parallel execution of such "speculative" OR-nodes [12] is continued. When a cut is executed, speculative computations are undone; otherwise if the cut is not executed, these computations are validated.

Interaction between cut and side effects Some branches of the search tree, under the scope of a cut, can therefore be computed in parallel executions while this would not have been the case sequentially. Side-effects performed in such branches ought to be undone. Therefore, side-effects executed in such conditions should be "reversible" either by restoring the previous state of computation, or by performing the side-effect in some cache that will be cleared if the side effect is not to be validated.

3.1. Maintenance of the sequential order

Solutions adopted in shared-memory systems [12, 2] cannot be used in PloSys. These solutions demand the exploration of the Prolog search tree to identify the left-most worker of the tree. This approach would require the use of a large number of messages in a distributed memory architecture. Instead, our approach is based on keeping the sequential order in the scheduler and using dedicated processes to control each side effect on each worker. This solution is not costly since the scheduler can keep track of the sequential execution order by computing the position of the workers in the search tree from the work transfers between workers. A list is used to point the relative positions of the workers in the program execution tree. When a worker exports work, it exports the highest (oldest) ones in the tree. Therefore the importer worker will be located at the right of the exporter and will be its successor in the list (see figure 3). Idle workers are removed from the list, until they benefit from new transfers. Although the list needs to be slightly complicated to support the management of side-effects, it is the basis for keeping the sequential order of side-effects (see section 3.4.).

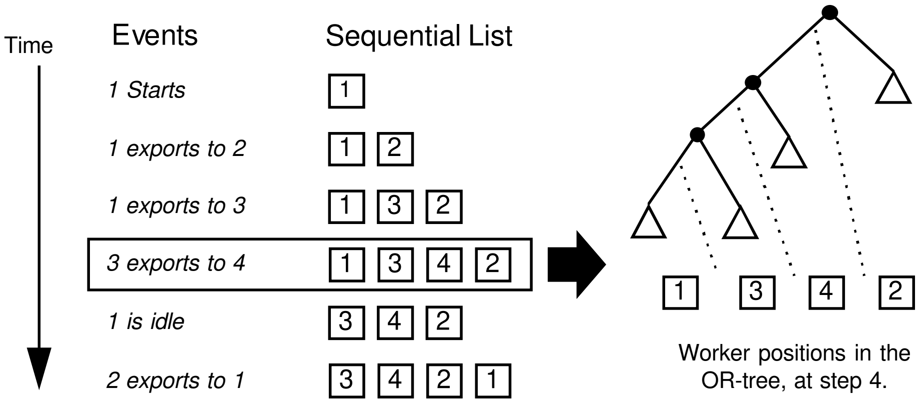

Figure 3: Maintenance of the sequential order

3.2. Identification of speculative work

The management of the cut by the workers include two phases. The first one, associated to compilation, is the maintenance of the knowledge of the speculative nature of some choice points. The second phase is the control of exported speculative choice-points and their associated work.

In a sequential system, a dedicated variable is used to store the top of the execution stack before calling a predicate including a cut. In an OR-parallel system, part of this stack portion may have been developed by another worker which needs to be identified to validate or unvalidate its work. Similarly, a worker importing choice points under the scope of a cut, needs to be informed of the "speculativity" of this work in order to manage correctly its side-effects. This requires a slight modification of the WAM instructions managing choice-points. A global variable indicates the current cut level during the execution and two fields are added to each choice point to store and restore this level (see figure 4). These modifications do not change the sequential behavior of the Prolog inference engine and the associated overhead remains very low.

3.3. Management of cuts

In order for the scheduler process to take cuts into account, it is sufficient to know the "speculativity level" of the available work, to limit as much as possible the number of messages. Only the load information message needs to be modified to represent speculative loads. Specific scheduling problems raised by the management of speculative work are addressed in section 3.5..

When several workers execute a program, exported choice points need to be marqued and their new (worker) owner memorised in order to be able to propagate the execution of a cut (validation or invalidation of work). An exported choice point will thus hold a special mark (such as a negative cut level, see figure 5) so that its previous "owner" can take appropriate decisions upon backtracking

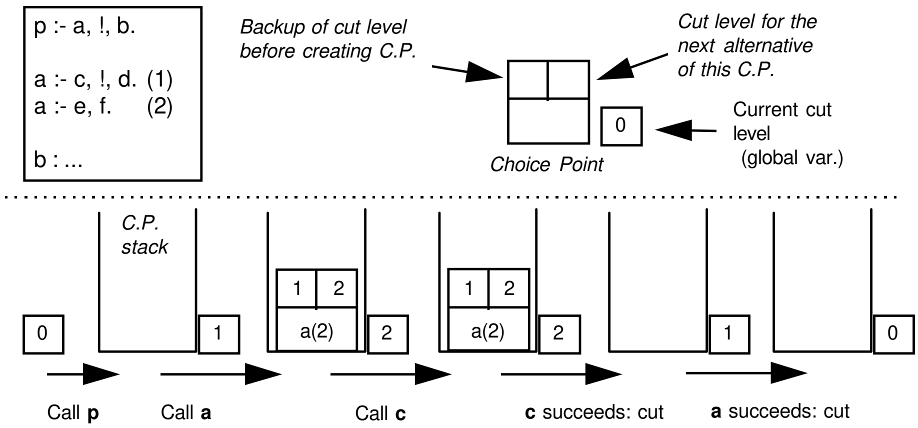

Figure 4: Cut management with no work exported

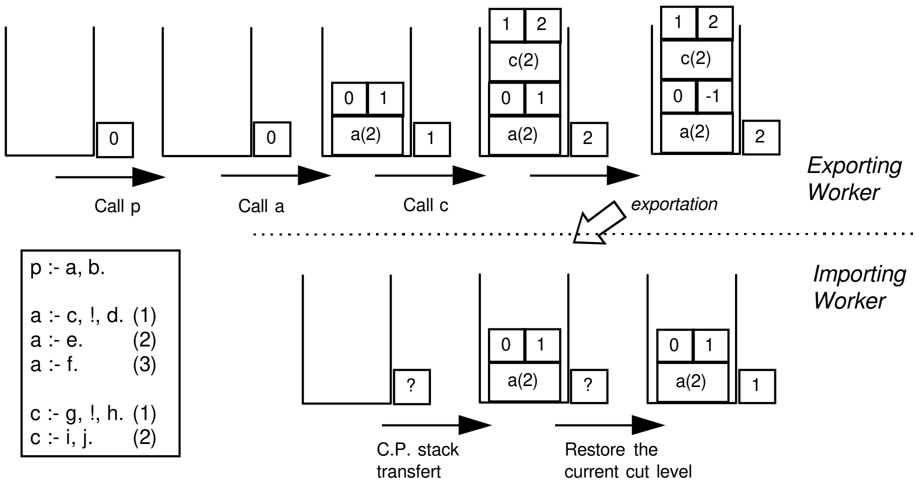

Figure 5: Management of cuts and work exportation

or executing a cut. The new "owner" identification needs also to be stored in a specific field of the choice point.

When a worker backtracks across an exported choice point, the cut management process of the worker (see figure 2), will check the choice point stack to identify all the workers developing branches issued from speculative choice points. A validation message has to be sent to each of these workers before the backtracking worker can resume its work. Similarly, if a worker executes a cut, its cut management process will send an invalidation message to each worker having imported work from one of the choice points concerned by the cut (see figure 6). It may occur that a worker has exported several choice points, previously imported or developed from previously imported ones. In this case if the exporter process receives a validation/invalidation message, it must forward it to the cut management process of the importer worker. Last, if a worker receives a validation message, it must update the cut levels stored in its choice points as well as its global cut level.

It is possible to avoid speculative work by using a "cavalier cut" [12] whose execution prunes all other solutions of a predicate. However, interactions with other side effects need to be taken into

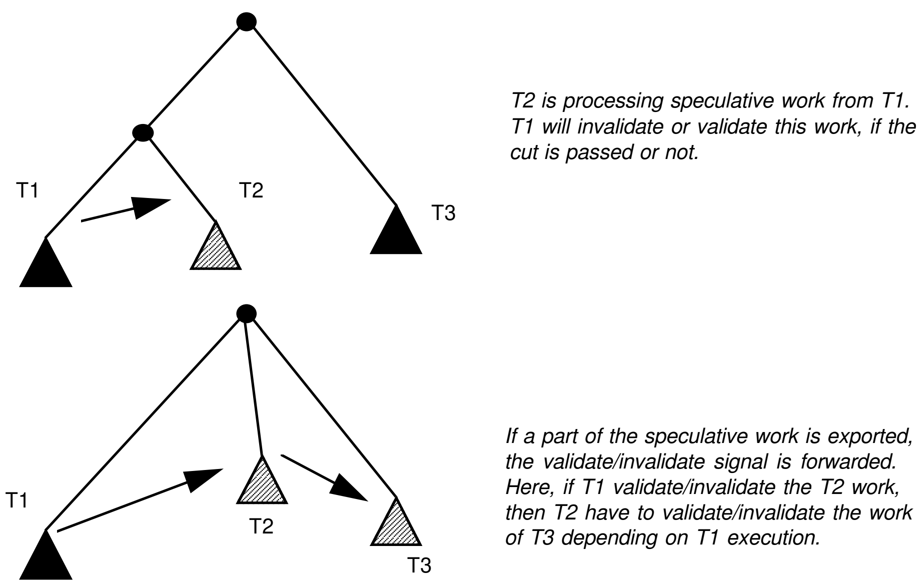

Figure 6: Cut messages propagation

account to avoid mixing two different solution flows (see [19] for example). Another possibility is to prohibit the exportation of choice points under the scope of a cut (see a simple possible solution in section 3.5.).

3.4. Inputs and Outputs management

The input case is the most simple since it is necessary, for a worker to read a data, to be left-most. The management of inputs thus amounts to a simple test to check whether the worker is left-most. If this is not the case, it will be suspended until it is notified by the scheduler that it has become left-most.

In PloSys, the inputs/outputs are managed by a dedicated server called “console” process. This process interacts between a specialised process within each worker and the outside world. A worker performs all its outputs in a local buffer. When the worker is the first one in the worker list of the scheduler (see section 3.1.), outputs are transfered to the “console” process and the buffer is reset. If a worker runs out of work and it has pending outputs, these must be preserved to be placed correctly in the output flow and respect the sequential Prolog order. The worker list of the scheduler includes a special marker, associated to each output buffer containing a pending output done by a worker before becoming inactive (see figure 7). When this marker becomes the first element of the list, the scheduler notifies the inputs/outputs management process of the associated worker to perform the output. One consequence of using buffered outputs is a reduction of the amount of communications to the “console” process.

3.5. Influence of side effects on the scheduling

The management of side-effects as described above, introduces new constraints on the scheduling. We describe here the main solutions but there remains still a lot of experimental work to be performed.

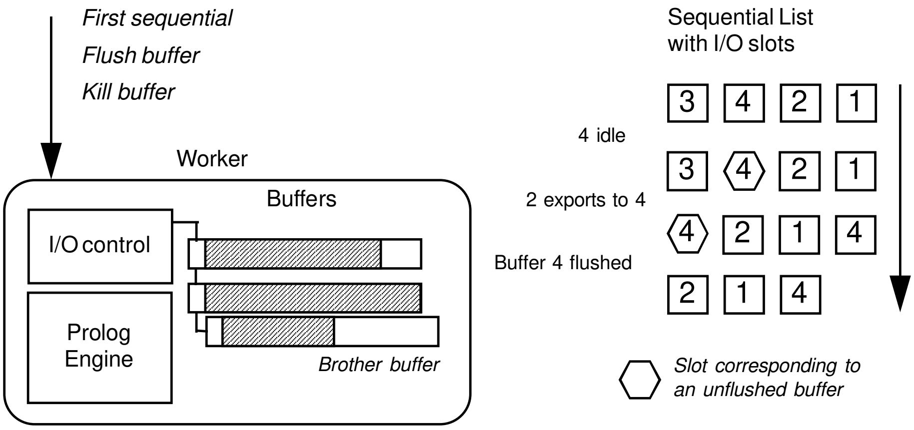

Figure 7: Management of inputs/outputs

Cut: there exist several possibilities to take into account speculative work in the scheduling [12]. The first is to make no distinction between speculative and non speculative work. Scheduling is then easier but this solution may be inefficient if a large number of workers execute speculative work which proves later unuseful. Another possiblity is to give a lower weight to speculative work in order to favor the dispatching of non-speculative work. The last possibility is to favor the allocation of work in the left part of the Prolog execution tree in order to reduce the amount of speculative work available.

Inputs/Outputs: when many workers are blocked, waiting for inputs, the scheduler should allocate work to the left of these workers. A similar situation may occur when the output buffer space of some workers is full.

4. Implementation and preliminary results

4.1. Prolog engine and parallel programming environment

A portable PloSys prototype was implemented using the *wamcc* [10] inference engine and the Athapascan [6] parallel programming library. The *wamcc* system includes a compiler which translates Prolog programs into Warren Abstract Machine (WAM) [23] instructions and then translates WAM instructions into C. The main interest of using *wamcc* is that the inference engine is not "too" big and can thus be "easily" modified, although the amount of modifications required to implement PloSys remains limited. Another interest of *wamcc* is that it represents a good compromise between performances and portability: its performances are close to the performances of the compilers generating native code while it only requires a GCC compiler to be ported to a new target processor[4]. *wamcc* is about 1.6 times faster than emulated SICStus 2.1 and 1.7 times slower than native code SICStus 2.1 [8].

PloSys uses also the Athapascan parallel programming library developed within the APACHE research project. In Athapascan, a parallel application is executed by a set of identical virtual processors executing asynchronously. Parallelism is triggered by blocking or non blocking remote procedure calls which hide the communication protocols used to transmit parameters and results.

[4] *wamcc* exploits the GCC possibility of generating *jumps* between functions to reduce the size of the stacks necessary to the computation.

4.2. Preliminary results

A PloSys prototype is partially running on workstation (SUN OS4), personnal computer (Linux) and distributed memory multiprocessor IBM SP-1.8. The benchmark results given in the sequel were obtained on this latter prototype. In the SP-1.8 prototype, inter-process communications use the Ethernet connection instead of the communication switch of the machine, for technical reasons which are out of the scope of this paper. In addition, cut and side-effects being currently tested, existing results were obtained with programs which do not use these instructions.

The overhead arising from the (limited) modifications done to *wamcc* to implement PloSys depend on the frequency of load transmission from the workers to the scheduler. It is limited to 1 % of the execution time of benchmarks if workers do not send any load information. However it may rise to 20 % if load information is transmitted every 0.1 s., because of the inefficiency of Ethernet communications and of the problems related to the use of Athapascan. In parallel, performances are good for "large" sized problems and when the number of processors is limited (see figure 8).

The use of Athapascan to implement PloSys raises a number of problems. Because of the lack of preemption and the impossibility to transmit signals, it is not possible to react rapidly to load variations and manage side-effects efficiently. Most of these problems are currently taken into account in the definition of a new version of Athapascan.

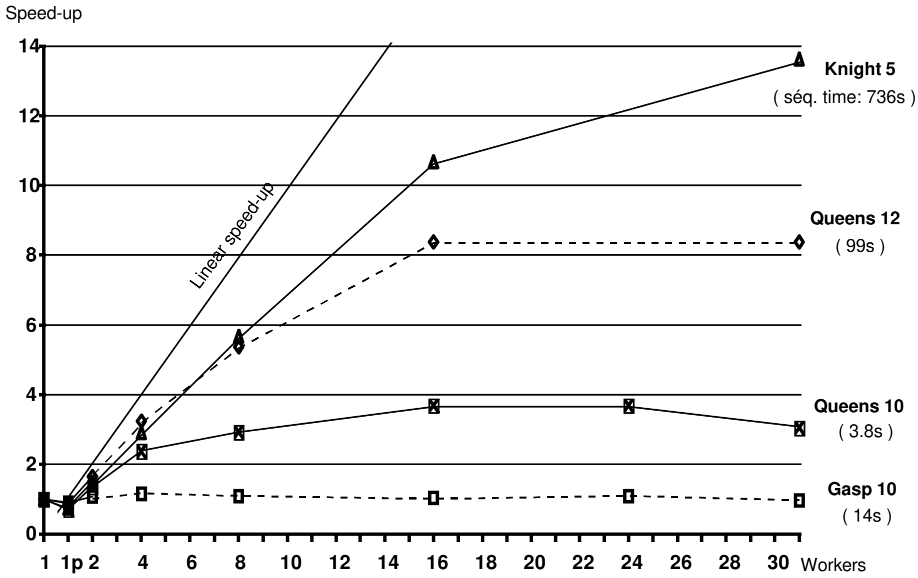

Figure 8: Benchmark results of the PloSys SP-1.8 prototype

Most benchmarks used in the past to measure OR-parallel logic systems (see for example [1]), cannot be used to test PloSys because of their "too" small size: for example, it only takes 110 milliseconds to compute the whole set of solutions to the classical eight queens problem, using *wamcc* running on a 133 Mhz Pentium processor. Defining a new set of benchmarks, better suited to the performances of modern micro-processors, was undertaken jointly by LMC-IMAG and KFKI and is currently in progress.

5. Related work

The PloSys system derives directly from the OPERA system [5]. Compared to OPERA, the computational model has been simplified since some OPERA optimisations, targeted to the Supernode parallel architecture used in OPERA, cannot be used in PloSys. This is the case in particular of the possibility for workers to carry on computing while exporting work. The most important difference with respect to OPERA is the management of side effects, integrated from the design of PloSys. Other differences with OPERA come from different design decisions which favor portability and modularity in PloSys, to make experimentations and evolutions easier. The OR-parallel system of the Malaga university [4] is also multi-sequential and implemented on a distributed memory computer. However it does not yet include any side-effect.

Several computational models avoid inter-workers communications, where each worker recomputes the entire path leading from the root of the Prolog search tree to its branch. Branch allocation to workers is done either by the application of a tree exploration rule by each worker [17], or by a centralised scheduler [7]. This type of model makes it difficult to implement Prolog cut and side-effects. One solution was proposed to adapt the cavalier cut to this type of model [13].

Muse [2] is an OR-parallel logic programming system based as PloSys on copying. However, the Muse system assumes some shared memory between workers, to implement the sharing of choice points. This shared memory area is also used to manage Prolog cut and side-effects. Unlike Muse, PloSys assumes that workers do not share any memory at all. Indeed it could be interesting to adapt Muse to distributed memory architectures, using some ideas of PloSys: the shared memory area would then be managed by the centralised scheduler, while accesses to this area would be done by message exchanges between the workers and the scheduler.

A lot of work was dedicated to the implementation of Prolog cut and side-effects in Aurora [18, 12] but all solutions assume the existence of a shared memory such as in a symmetric shared memory multiprocessor including a limited number of processors. The techniques developed for Aurora seem difficult to adapt to the type of parallel systems aimed by PloSys.

6. Conclusion and future work

PloSys is an OR parallel logic programming system offering most Prolog functionalities on distributed memory parallel systems. The programming model includes a novel solution to implement cut and side effects on this type of parallel system. A PloSys prototype was implemented on network of personal computers and workstation as well as on a IBM SP-1.8.

In the short term, tuning of the prototype will be continued, in particular to test the cut and side-effects implementation and their relation with the scheduling policies. In the longer term, the prototype will be entirely distributed. The distributed scheduling algorithm will use some of the techniques developed in the LOGFLOW system [21, 14] for which there already exists an animation tool [20] that can be used to tune load-balancing strategies. Another foreseen extension is to combine constraint propagation techniques, already available in the inference engine *wamcc* [11] with OR-parallelism. Such a combination was already proven to be very effective [22].

Acknowledgements

All Apache project research reports are available by *ftp* at *ftp.imag.fr* in the directory *pub/APACHE/RAPPORTS*.

References

[1] K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In Saumya Debray and Manuel Hermenegildo, editors, *Proc. of the 1990 North American Conference on Logic Programming*. MIT Press, 1990.

[2] K.A.M. Ali and R. Karlsson. Full prolog and scheduling or-parallelism in muse. *International Journal of Parallel Programming*, 19(2):445–475, december 1990.

[3] L. Araujo and J.J. Ruz. PDP: Prolog Distributed Processor for Independent_AND\OR Parallel Execution of Prolog. In *Proc. of ICLP'94, 11th International Conference on Logic Programming*, pages 142–156. The MIT Press, 1994.

[4] V. Benjumea and J.M. Troya. An or parallel prolog model for distributed memory systems. In *Programming Language Implementation and Logic Programming*, LNCS 714. Springer Verlag, August 1993.

[5] J. Briat, M. Favre, C. Geyer, and J. Chassin de Kergommeaux. OPERA: OR-Parallel Prolog System on Supernode. In Kacsuk and Wise [15], chapter 3, pages 45–63.

[6] M. Christaller. Athapascan-0a sur PVM3: définition et mode d'emploi. Rapport APACHE 11, IMAG, Grenoble, February 1994. Disponible sous `ftp.imag.fr:imag/APACHE/RAPPORTS`.

[7] W.F. Clocksin and H. Alshawi. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. *New Generation Computing*, 5:361–376, 1988.

[8] P. Codognet and D. Diaz. WAMCC: Compiling Prolog to C. In *Proc. of ICLP'95, 12th International Conference on Logic Programming*, Tokyo, Japan, 1995. The MIT Press.

[9] J. Chassin de Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, september 1994.

[10] D. Diaz. wamcc prolog user's manual. Rr inria, INRIA-Rocquencourt, Domaine de Voluceau, 78153 Le Chesnay, July 1994.

[11] D. Diaz and P. Codognet. A minimal extension of the wam for clp(fd). In *Proc. of ICLP'93, 10th International Conference on Logic Programming*, Budapest, Hungary, 1993. The MIT Press.

[12] B. Hausman. *Pruning and Speculative Work in OR-Parallel Prolog*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 1990. SICS research report D-90-9001.

[13] P. Kacsuk. Or-parallel prolog on distributed memory systems. In *Proc. of PARLE'94*. Springer-Verlag, 1994.

[14] P. Kacsuk. Wavefront scheduling in logflow. In *Proc. of the Euromicro Workshop on Parallel and Distributed Processing*, pages 503–510. IEEE Computer Society Press, 1994.

[15] Peter Kacsuk and Michael Wise, editors. *Implementations of Distributed Prolog*. John Wiley and Sons, 1992.

[16] S.E. Kannat, E. Morel, J.P. Kitajima, and J. Briat. A platform to study dynamic load balancing functions for parallel logic systems. In *IEEE First International Workshop on Parallel Processing*, Bangalore - India, 1994.

[17] Z. Lin. A distributed fair polling scheme applied to or-parallel logic programming. *International Journal of Parallel Programming*, 20(4), August 1992.

[18] E. Lusk et al. The Aurora Or-Parallel Prolog System. In ICOT, editor, *Proc. of the International Conference on Fifth Generation Computer Systems 1988.* ICOT, 1988.

[19] E.L. Lusk, S. Mudambi, Overbeek R, and P. Szeredi. Applications of the aurora parallel prolog system to computational molecular biology. In *Proc. of the JICSLP'92 Post-Conference Joint Workshop on Distributed and Parallel Implementations of Logic Programming Systems*, Washington DC, November 1992.

[20] Zs. Nemeth and P. Kacsuk. Performance measuring facilities in logflow. In *Proc. of European School on Parallel Programming Environments, ESPPE'96*, Alpe d'Huez, 1996.

[21] Zs. Puskas and P. Kacsuk. Distributed token scheduler for the ws-logflow. submitted to DAPSYS'96.

[22] P. van Hentenryck. Parallel Constraint Satisfaction in Logic Programming: Preliminary Results of Chip within PEPSys. In *Proc. of the Sixth International Conference on Logic Programming*, pages 165–180. MIT Press, 1989.

[23] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report tn309, SRI, October 1983.