Or-Parallel Scheduling Strategies Revisited

Inês de Castro Dutra[1] and Adriana Marino Carrusca[2]*

[1] COPPE/Departamento de Engenharia de Sistemas e Computação
Universidade Federal do Rio de Janeiro, Brasil
ines@cos.ufrj.br
[2] IBGE/RJ, Brasil
marino@ibge.gov.br

Abstract—
Parallel logic programming systems have been studied for more than a decade. Techniques and scheduling strategies for or-parallel systems have been established for systems that run on centralised memory architectures. As new parallel platforms such as clusters of workstations or clusters of PCs gain popularity, these techniques vastly studied for centralised memory systems may become obsolete and inefficient. In this work we study several scheduling strategies commonly used in or-parallel systems designed for centralised memory architetcures. We simulate these strategies on different parallel environments in order to estimate the costs associated to these scheduling strategies in each environment. We use a benchmark set commonly studied by the or-parallel community. Our study concentrates on simulating top-most, bottom-most, and left-most strategies while modelling costs associated to centralised and distributed memory architectures. We then implement our own strategy that selects best work based on the costs to move to a piece of work. Our results show that depending on the characteristics of the search tree none of the three strategies: top, bottom or left-most can yield good results. Our proposed strategy based on taking minor costs tasks proved to produce better results than any of these fixed strategies, particularly for distributed-memory platforms.

Keywords— **or-parallelism, or-parallel scheduling, logic programming, parallel architectures, simulation**

I. Introduction

Parallel logic programming systems have been studied for more than a decade. Techniques and scheduling strategies for or-parallel systems have been established [1], [2] for or-parallel systems that run on centralised memory architectures. Some work has been done on distributed memory systems such as Opera [4], Plosys [15], [9], [7], [6] Delphi [8], and Kabu-wake [10]. More recently, Gupta et al proposed a stack-splitting technique to distribute work in ACE [12], [20]. Most of these strategies concentrates on one kind of policy based on top-most, bottom-most or left-most dispatching, and, in fact, there is no work in the literature that compares the impact of these strategies in the same or-parallel system. Beaumont [3] performed some work in this direction, but his main concern was to devise a flexible strategy to Aurora [14].

These fixed strategies, top-most, bottom-most and left-most, seem to be efficient for most or-parallel systems running on centralised memory or distributed memory architectures, but as new platforms such as clusters of workstations and clusters of PCs have been gaining popularity, these strategies may become obsolete or inefficient.

In this work we simulate top-most, bottom-most, and left-most based strategies modelling their associated costs in different architetcures: centralised and distributed memory systems. We evaluate each strategy for a benchmark set usually employed to study or-parallel systems, modelling the costs to distribute work in each platform. Based on the results of this study we also simulate a strategy that chooses the best piece of work according to the costs modelled.

Our results show that, in general, top-most dispatching performs better than the other strategies for a greater number of processors. Our proposed strategy based on minor cost tasks is overall better, particularly when we introduce architectural costs to the simulation.

Other works have been done on simulation of parallel logic programming scheduling strategies. Shen has simulated the Argonne scheduler for Aurora [16], and did detailed simulation of and-or parallelism in order to obtain amount of parallelism from applications for infinite and finite number of processors [17]. Fernández worked in the same track by doing a study of and-parallelism in applications [11]. Our work differs from theirs in that we simulate different scheduling strategies modelling costs for centralised and distributed memory architectures. Work was also done in actual parallel systems such as Aurora [3], as mentioned before, where Beaumont experimented with a combination of strategies in the Bristol scheduler, for Aurora, on a shared-memory architecture.

The paper is organised as follows. Section III describes our simulator, and the fixed scheduling strategies implemented. Section IV explains our modelling for costs. Section V presents a brief description of our benchmark set. Section VI presents our results. Finally last section draws our main conclusions and future work.

II. Or-Parallel Execution

The execution of logic programs generates execution trees where each branch corresponds to an alternative path to solve a given goal in the body of a logic program clause. Or-parallel execution assumes that each alternative for a goal can be executed by a different processor. In that case, if we

*This research was partially sponsored by CNPq, Brazilian Research Council

```
paper(C,T,A,D,L) :- conference(C), title(T), author(A), date(D), location(L).
paper(T,L) :- tech_rep(T), location(L).
conference(SBAC) :- ...
conference(PDP) :- ...
conference(ISCA) :- ...
conference(ASPLOS) :- ...
```

Fig. 1. Example of or-parallelism

have a program such as the one shown in figure II, and we have available processors we could execute each alternative to the goal `conference/1` by a different processor. As each clause leads to a different solution, each computation, in this case, can be done independently. Due to lack of space, a discussion of or-parallel models is out of the scope of this paper. According to kind of application and to the total number of available processors, different scheduling strategies may produce different execution trees.

III. Simulator

We studied three basic strategies commonly used in parallel logic programming systems that exploit or-parallelism: left-most dispatching, bottom-most dispatching, and top-most dispatching. When dispatching on left-most work, the scheduler selects the tasks that were created in the program clause order. This strategy assumes that programmers place solutions in the program clause order. This also favours execution of non-speculative work. When dispatching on bottom-most tasks, the scheduler selects work that is deeper in the computation tree. This strategy is normally used to maintain processor locality and to minimise contention on shared choicepoints. When selecting top-most tasks, the scheduler chooses work that is higher in the tree, i.e, closest to the root. This strategy was originally used by the first or-schedulers implemented for centralised-memory systems such as Aurora [14]. The heuristics behind top-most dispatching is that higher tasks have greater granularity, therefore processors would communicate less often.

Our simulator implements these three strategies ideally and considering costs of two different machine architectures: centralised-memory and distributed-memory systems.

We also simulated a strategy that always selects the closest task to a processor, instead of using a fixed left-most, top-most or bottom-most policy. In a real system, this information can be kept in a data structure such as the **tip nodes** employed by Sindaha in his Dharma scheduler [19]. In the tip nodes scheme it is very likely that the closest task to a processor is the one that is found on its left or on its right in the tip list. Therefore, the minor cost strategy implemented this way would yield very low overheads.

Our simulator executes Prolog programs in parallel in a step-by-step fashion. We keep a global clock to synchronise all processors. Each clock cycle corresponds to one Prolog reduction (matching of a goal with a candidate clause). We also maintain a local clock per processor. The main algorithm of the simulator is roughly sketched in figure 2. Each processor tries to find an alternative clause to its current goal. All other alternatives to the same goal are put on a task queue. Processors can be in one of three possible states: **idle**, **busy** or **waiting**. All processors, but the first one, are initialised with idle state. They change to busy when the scheduler finds an available task to be executed. Processors enters the **waiting** state when we model architectural costs and the processor selects a task from the task queue. Instead of executing the task immediately, the processor simulates its waiting time depending on the architecture.

We implemented three versions of the simulator:

- **ideal**, simulates each Prolog reduction per processor, without considering any cost or delay per processor. All processors are allowed to execute one Prolog reduction at a global cycle;
- **task cost**, in this version we compute the *distance* each processor needs to travel to arrive to the selected task from the task queue;
- **task cost with processor delay**, in this version whenever the scheduler assigns a task from the task queue to a processor, a timer is started to that processor and its state is changed to waiting. The value of the timer depends on the cost to take the task chosen and also on the architecture modelled.

In order to implement the top-most and bottom-most dispatching strategies we rank the task queue according to the level of the task in the execution tree. In order to implement the left-most dispatching strategy we keep an identifier for each choicepoint node that is used by the algorithm to insert its descendants tasks at the right position in the task queue. Carrusca presents a more detailed explanation about the simulator in her thesis [5].

IV. Modelling Costs

Our first implementation, **ideal**, does not consider any costs in order to compute execution time. Therefore, this version provides a task-processor schedule that is ideal to a given application, i.e., all tasks have the same probability to be taken.

In the ideal simulation (**ideal**), every idle processor can take any task in the same clock cycle, and the cost to take a task is equivalent to one clock cycle.

```
While not all processors idle do
   for each busy processor do
find an alternative to its current goal
        increment clock[i]
if leaf found then
            processor[i] = idle
            produce solution
        else
          if there is no alternative goal
            processor[i] = idle
            produce a fail
          else /* an alternative was found */
              point registers to the alternative body
          endif
        endif
        find other alternatives, if any, and put them on the task queue
   end for
   Schedule tasks from the task queue to idle processors
   Increment global clock
end while
```

Fig. 2. Simulation Algorithm

In the simulation with costs (**task cost**), we increase the idle processor's clock with the **task cost**. The task cost in this implementation is defined by the distance between the last task the processor was positioned at and the task to be executed. By distance we mean the cost of traversing the search tree until the processor reaches the choicepoint with alternatives left to be executed. This cost depends on the scheduling strategy employed. In an actual Prolog implementation, this is directly related to de-installing and installing variable bindings along the path through the Prolog stacks in order to reach the selected choicepoint to take an alternative from.

In the simulation with processor delay (**task cost with processor delay**), whenever a processor takes a task, a timer is initiated depending on the modelled architecture, and the processor enters the waiting state. If the simulated architecture has centralised memory, the timer is started with the value of the task cost calculated as in the **task cost** simulation.

If the simulated architecture has distributed memory, the timer is initialised with the value shown in expression 3: where num_bytes represents the number of bytes of each choicepoint to be traversed, msg_size corresponds to the size of the network message, and $network\_latency$ represents the delay to send a message through the network. This expression represents the cost to traverse the tree to find a task plus communication costs. In our implementation we considered that each choicepoint has 120 bytes, a message has 4 bytes, and the latency is equivalent to half a Prolog reduction.

As in our simulation all tasks are placed on a global task queue, there are some tasks that are taken by the same processor that created them. In that case, no communication cost is added to that processor in the **task cost with processor delay** simulation, with distributed memory.

V. Benchmark Set

Our benchmark set is composed of four programs that benefit from or-parallelism. The first one, `family`, was used only to validate our results and consists of a database with family and employment relationships. The query used for this program is replicated in order to produce a choicepoint with 35 alternatives in the beginning of the computation. Our second application is the popular N-queens program (`queens`). We ran this program with a 6x6 chessboard. Our third program consists of a theorem prover, `mutest` that solves a problem described in [13]. The problem is to produce the string 'mu' from a given string, through production rules that can be applied in any order. The theorem to be proved is that 'miiiii' produces 'mu'. If we replace the letters in the strings by numbers, this puzzle solves some interesting problems in number theory. Our last program is an example from the well-known natural language question-answering system `chat-80`, written at the University of Edinburgh by Pereira and Warren. This version of `chat-80` operates on the domain of world geography. The program `chat80` makes queries to the `chat-80` database. This is a small scale benchmark with good or-parallelism, and it has been traditionally used as one of the or-parallel benchmarks for both the Aurora and Muse [1] systems.

We chose benchmarks usually used by parallel logic pro-

$$task\_cost + \frac{(task\_cost \times num\_bytes)}{msg\_size} \times network\_latency$$

Fig. 3. Cost expression

gramming systems that exploit or-parallelism. These programs have different search trees in order to stress the behaviour of different scheduling policies.

VI. RESULTS

We executed the three versions of the simulator with 1, 2, 4, 8, 16, and 32 processors. The run times obtained for one processor do not consider any parallel costs. All speedups shown in the graphs and tables are relative to these sequential run times.

Figures 4, 5, 6, and 7 show the speedups of our benchmarks up to 32 processors, for the three policies studied. From left to right, the bars show speedups for the top-most, bottom-most, left-most and minor-cost policies, respectively.

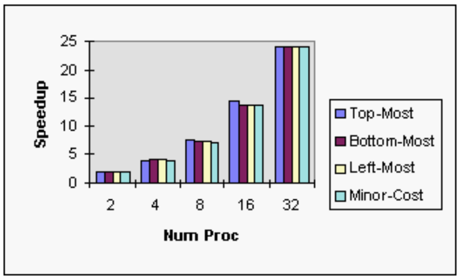

Fig. 4. `family` speedups, ideal

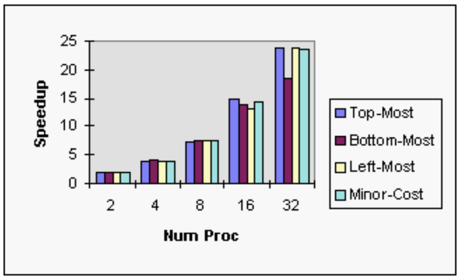

Fig. 5. `mutest` speedups, ideal

The ideal simulation shows the maximum speedup we can achieve up to 32 processors for all applications and for each strategy. We can observe that although no cost is associated to dispatching work to processors, the applications are affected by different policies. This indicates that the search trees have different characteristics and their balance can be good or bad depending on the policy used. Tables I, II, III, and IV show speedups for all applications with 2, 4, 8, 16 and 32 processors, for all policies. We can observe that for `mutest` and `queens`, as we increase the number of processors, the performance improves for the top-most dispatching strategy. This is mainly due to the characteristics of the search tree of these two applications that create choicepoints that have more granularity closer to the root. In that case, branches of the tree that are on the critical path are taken earlier producing better performance.

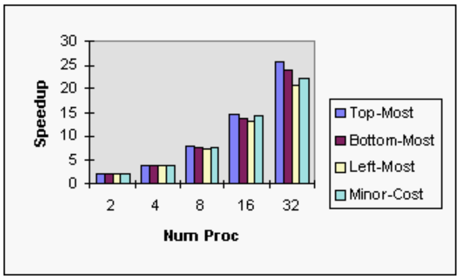

Fig. 6. `queens` speedups, ideal

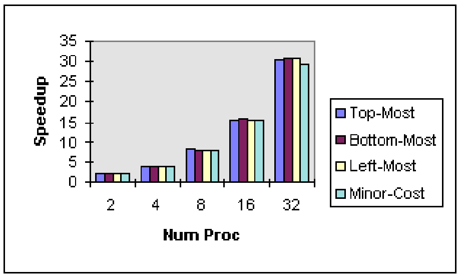

Fig. 7. `chat-80` speedups, ideal

The top-most policy is 7.6% better than bottom-most, for the `queens` benchmark, with 32 processors, and is 24.8% better than left-most for the same example. The best improvement is achieved when we compare the top-most policy with the left-most policy, with 32 processors, for the `mutest` application: top-most is 29.4% better than left-most. As the `chat-80` application has a very good amount of parallelism, no ideal simulated strategy produced visible differences in performance.

TABLE I

family -MAIN - IDEAL

Num Proc	Top Most	Bottom Most	Left Most	Minor Cost
2	1.96	1.99	1.99	1.94
4	3.90	3.96	3.96	3.90
8	7.60	7.41	7.41	7.22
16	**14.45**	13.76	13.76	13,76
32	24.08	24.08	24.08	24.08

TABLE II

mutest - MU_TOP - IDEAL

Num Proc	Top Most	Bottom Most	Left Most	Minor Cost
2	1.99	1.98	1.99	1.98
4	3.91	3.96	3.84	3.91
8	7.28	7.69	7.48	7.62
16	**14.82**	13.8	13.13	14.29
32	**23.75**	18.35	23.75	23.40

TABLE III

queens - QUEENS(6,QS) - IDEAL

Num Proc	Top Most	Bottom Most	Left Most	Minor Cost
2	1.99	1.99	1.99	1.99
4	3.94	3.89	3.85	3.94
8	7.78	7.56	7.27	7.60
16	**14.52**	13.73	12.93	14.26
32	**25.62**	23.80	20.52	22.22

When we introduce costs to take the tasks (**task cost**), i.e., the processor takes a task, but its clock is increased of the cost to take the task (distance from the node where the processor is positioned to the task node), we can observe a decrease in the speedups when compared with the ideal simulation results, as expected. We also obtain slowdown for some numbers of processors. Note that we compare the parallel execution times with the sequential execution time of an interpreter that does not consider any parallel overheads, therefore these speedups are somewhat worse than if we were comparing with a sequential version that has parallel overheads.

Figures 8, 9, 10, and 11 show the speedups for the task cost simulation for all applications, with the three policies. In

TABLE IV

chat-80 - IDEAL

Num Proc	Top Most	Bottom Most	Left Most	Minor Cost
2	1.99	1.99	1.99	1.98
4	3.99	3.99	3.99	3.96
8	7.96	7.95	7.95	7.91
16	15.67	15.77	15.70	15.47
32	30.41	30.67	30.67	29.29

general, for a small number of processors, the top-most policy performs worse than the left-most and bottom-most policies. As we increase the number of processors, the top-most policy performs better than the bottom-most and left-most policies. The best improvement is observed for the chat-80 application, where the top-most strategy is 44% better than the left-most, with 32 processors.

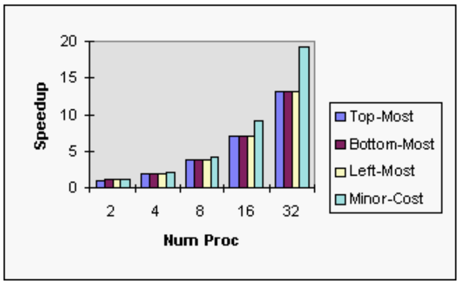

Fig. 8. family speedups, task cost

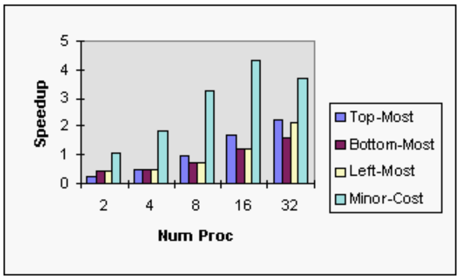

Fig. 9. mutest speedups, task cost

Our proposed strategy, that allocates minor tasks to processors, performs relatively well for the ideal simulation, and, when we introduce costs, performs much better than any other strategy as we increase the number of processors. The only exception is queens, with 32 processors. This can be better observed in table V. In this case, top-most shows better

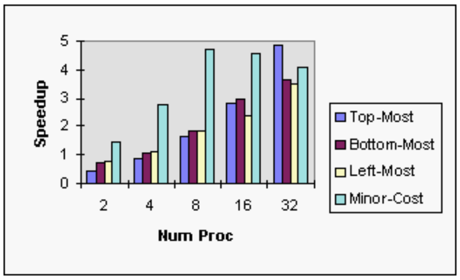

Fig. 10. `queens` speedups, task cost

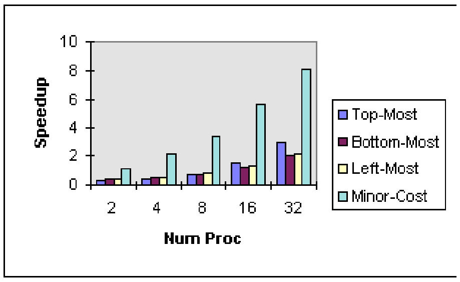

Fig. 11. `chat-80` speedups, task cost

performance, because the top-most strategy potentially starts nodes that are in the critical path first, while our minor cost strategy can select bottom-most nodes that may not be in the critical path. This may lead to delaying the execution of tasks that are in the critical path.

Our minor cost strategy can be a good alternative to the other strategies as we increase the number of processors, because in that case, processors that are taking a node for the first time will be allocated to tasks that are closer to the root. Therefore this strategy favours both the expansion of nodes that are closer to the root and the expansion of bottom-most nodes to keep locality.

TABLE V
`queens` - QUEENS(6,QS) - TASK COST

Num Proc	Top Most	Bottom Most	Left Most	Minor Cost
2	0.47	0.71	0.76	1.47
4	0.86	1.05	1.13	2.80
8	1.68	1.88	1.89	4.68
16	2.81	2.98	2.37	4.56
32	4.84	3.62	3.47	4.10

When we introduce architectural costs and delay processors, the behaviour of the strategies contrasts with the ideal or task cost simulations. The distributed memory simulation presents slowdown for almost all applications and strategies (note again that the sequential times used to compute speedups do not consider parallel costs. Results would be somewhat better if we used the time obtained by the parallel simulation in one processor). Although we have slowdown for almost all applications with almost all numbers of processors, we can observe that as we increase the number of processors, the rate between any other strategy (except minor cost) and the top-most strategy reduces until top-most becomes better. The only exception is `chat-80` that produces better results wih the left-most strategy. This can be observed in tables VI, VII, VIII, and IX.

TABLE VI
`family` - MAIN - TASK COST WITH PROCESSOR DELAY - DM

Num Proc	Top Most	Bottom Most	Left Most	Minor Cost
2	0.34	0.36	0.38	0.36
4	0.40	0.41	0.44	0.54
8	0.61	0.55	0.63	0.82
16	1.08	0.98	0.98	1.44
32	1.96	1.96	1.96	3.17

TABLE VII
`mutest` - MU-TOP - TASK COST WITH PROCESSOR DELAY - DM

Num Proc	Top Most	Bottom Most	Left Most	Minor Cost
2	0.03	0.28	0.33	0.33
4	0.05	0.27	0.22	0.40
8	0.08	0.20	0.19	0.36
16	0.17	0.28	0.35	0.35
32	0.32	0.31	0.28	0.28

TABLE VIII
`queens` - QUEENS(6,QS) - TASK COST WITH PROCESSOR DELAY - DM

Num Proc	Top Most	Bottom Most	Left Most	Minor Cost
2	0.06	0.36	0.66	0.51
4	0.10	0.27	0.34	0.66
8	0.17	0.35	0.36	0.93
16	0.36	0.39	0.44	1.13
32	0.70	0.57	0.59	0.91

Our proposed strategy that selects minor tasks also behaves quite well for greater number of processors. Particularly for our simulation of centralised memory (SM), this

TABLE IX

`chat-80` - TASK COST WITH PROCESSOR DELAY - DM

Num Proc	Top Most	Bottom Most	Left Most	Minor Cost
2	0.06	0.29	0.38	0.67
4	0.04	0.24	0.29	1.2
8	0.46	0.24	0.27	1.26
16	0.12	0.33	0.32	1.42
32	0.23	0.43	0.43	1.29

strategy performed 35% better than the other strategies, with 32 processors.

Our proposed strategy tries to allocate new idle processors to tasks that are closer to the root in order to increase work granularity, and to force processors to start as soon as possible new nodes that may be in the critical path. Therefore, its performance should be better than the other strategies as it mixes bottom-most dispatching for processors that become idle and top-most dispatching for processors that are new in taking work from the execution tree.

According to our experiments, the top-most strategy should be used instead of left-most or bottom-most strategies. In fact, to the best of our knowledge, no work has proved or compared top-most with bottom-most and left-most strategies in a fair way, in order to evaluate their impact in performance. Implementors of shared-memory or-parallel systems argue that bottom-most dispatching can produce better performance than top-most dispatching, because they maintain locality of references to the shared data. In this work we experimented with these strategies under the same conditions and for almost all applications top-most seemed to produce better performance. Our minor cost strategy that combines top-most with bottom-most can produce even better performance.

VII. Conclusions and Future Work

We implemented a simulator for the or-parallel execution of Prolog that executes four different scheduling strategies. We observed from our experiments that, contrary to results found in the literature, some applications benefit from top-most scheduling strategies. Our minor cost proposed strategy works as if new idle processors take work from nodes closest to the root, and forces processors that became idle to take work from the nearest node. This strategy has been proposed before [18], but to best of our knowledge was never implemented.

We believe that such kind of strategy would behave best in heterogenous systems where we have distributed-shared memory architectures such as clusters of PCs. Work is under way to do a detailed simulation of such an environment in order to test more benchmarks and devise better scheduling algorithms.

References

[1] Khayri A. M. Ali and Roland Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[2] Anthony Beaumont. *Scheduling in Or-Parallel Prolog Systems*. PhD thesis, University of Bristol, Department of Computer Science, In preparation, 1993.

[3] Anthony Beaumont, S. Muthu Raman, and Péter Szeredi. Flexible Scheduling of Or-Parallelism in Aurora: The Bristol Scheduler. In Aarts, E. H. L. and van Leeuwen, J. and Rem, M., editor, *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 403–420. Springer Verlag, June 1991. Lecture Notes in Computer Science 506.

[4] J. Briat, M. Favre, C. Geyer, and J. Chassin. Scheduling of Or-parallel Prolog on a Scalable, Reconfigurable, Distributed-Memory Multiprocessor. In *Proceedings of Parallel Architecture and Languages Europe*. Springer Verlag, 1991.

[5] Adriana Marino Carrusca. Estudo de Estratégias de Escalonamento para Andorra-I. Master's thesis, COPPE/Sistemas, UFRJ, MSc. thesis, in preparation, 1999.

[6] A. P. Centeno and C. Geyer. Penelope – Um Modelo de Escalonador Hierárquico para o Sistemas Plosys. In *X Simpósio Brasileiro de Arquitetura de Computadores, SBAC-PAD*, Setembro 1998.

[7] Ana Paula Bluhm Centeno. Penelope – Um Modelo de Escalonador Hierárquico para o Sistemas Plosys. Master's thesis, Universidade Federal do Rio Grande do Sul, Instituto de Informática, Curso de Pós-Graduação em Ciência da Computação, 1999.

[8] William Clocksin. Principles of the DelPhi Parallel Inference Machine. *Computer Journal*, 30(5):386–392, 1987.

[9] É. MOREL and J. BRIAT and Chassin de KERGOMMEAUX and C. GEYER. *Parallelism and Implementation of Logic and Constraint Logic Programming*, chapter Side-effects in PloSys OR-parallel Prolog on Distributed Memory Machines. Nova Science, Inc., 1999.

[10] K. Kumon *et al.* Kabu-Wake: A New Parallel Method and Its Evaluation. In *Proceedings of CompCon 86*, pages 168–172, 1986.

[11] M. J. Fernańdez, M. Carro, and M. V. Hermenegildo. IDRA (IDeal Resource Allocation): A Tool for Computing Ideal Speedups. In *ICLP'94 Pre-Conference Workshop on Parallel and Data-Parallel Execution of Logic Languages*, Facultad de Informática, Universidad Politécnica de Madrid, June 1994.

[12] Gopal Gupta and Enrico Pontelli. Stack-Splitting: A Simple Technique for Implementing Or-Parallelism and And-Parallelism on Distributed Machines. In *Proceedings of the 1999 International Conference on Logic Programming*, 1999.

[13] Douglas R. Hofstadter. *Gödel, Escher, Bach: an eternal golden braid*. Harmondsworth: Penguin, 1980.

[14] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2,3):243–271, 1990.

[15] E. Morel, J. Briat, J. Chassin de Kergommeaux, and C. Geyer. Side-Effects in PloSys OR-parallel Prolog on Distributed Memory Machines. In *JICSLP'96 Post-Conference Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages, Bonn, Germany*, September 1996.

[16] Kish Shen. An Investigation of the Argonne Model of Or-Parallel Prolog. Master's thesis, University of Manchester, 1986.

[17] Kish Shen. *Studies of And/Or Parallelism in Prolog*. PhD thesis, Computer Laboratory, University of Cambridge, 1992.

[18] Fernando M. A. Silva. *Implementations of Logic Programming Systems*, chapter Or-Parallelism on Distributed Shared Memory Architectures. Kluwer Academic Pub., 1994.

[19] Raéd Yousef Sindaha. Branch-Level Scheduling in Aurora: The Dharma Scheduler. In *Proceedings of the 1993 International Logic Programming Symposium*, pages 403–419, October 1993.

[20] K. Villaverde, H. Guo, E. Pontelli, and G. Gupta. Incremental Stack-Splitting: A Simple Technique for Implementing Or-Parallelism and

And-Parallelism on Distributed Machines. In *Proceedings of the Workshop on Parallelism and Implementation Technologies for (Constraint) Logic Programming Languages*, 2000.