

A Dynamic Task Distribution and Engine Allocation Strategy for Distributed Execution of Logic Programs

George Xirogiannis, Hamish Taylor

Dept. of Computing & Elec. Engineering
Heriot-Watt University
Edinburgh, EH14 4AS, Scotland, UK
G.Xirogiannis@hw.ac.uk, hamish@cee.hw.ac.uk

Abstract. Distributed execution of logic programs on heterogeneous processors requires efficient task distribution and engine synchronization to exploit the potential for performance. This paper presents a task-driven scheduling technique to distribute tasks to engines effectively. It consists of a dynamic hierarchy of distributed scheduling components able to adapt to program characteristics and the platform configuration and to control the considerable communication costs while exploiting good degrees of parallelism. It also incorporates an abort & failure mechanism to reduce speculative work and keep engines as busy as possible. Several experimental results illustrate the performance of the model.

1 Introduction

Logic Programming languages like Prolog support a form of programming where one declares the logic of the problem and the implementation provides the necessary control for efficient execution. Logic programs can take advantage of recent parallel and distributed architectures by exploiting parallelism mainly in the forms of divide-and-conquer and speculative execution. The distributed execution of logic programs on heterogeneous processors such as a LAN of workstations may create parallel tasks that need to be assigned to remote processors at run-time. A traditional task scheduler [16] relies heavily on shared resources to perform its functions. But performance will not increase proportionally if the scheduler operates on resources shared over a LAN of distributed workstations with considerable communication costs. Such cases require techniques that are better adapted to the nature of distributed computing.

This paper presents a scheme for effectively distributing tasks to engines on a process-based parallel logic programming system running in a distributed manner on the nodes of a virtual multiprocessor. The **Prolog Area Network** system [15] runs on a LAN of workstations with each Prolog engine running on a different workstation. Engines employ the services of PVM to communicate with each other either synchronously or asynchronously using extra message passing primitives added to SICStus Prolog. PAN is able to exploit various forms of parallelism

(AND, OR, combinations). Some of its particular merits are its robustness, ease of use and its ability to exploit highly available hardware. But the communication overheads of the distributed platform are significant dictating that any task distribution mechanism should add proportionally little execution overhead.

The following sections analyze the design choices of this model further. Section 2 briefly discusses other scheduling models and indicates existing pitfalls. Section 3 discusses the scheduling scheme used in PAN, presents the scheduling mechanism and analyzes its characteristics. Section 4 discusses the advantages of the proposed model in comparison to other techniques and finally section 5 presents some experimental results that illustrate the performance of the model.

2 Relevant Research

At the abstract level we can classify scheduling methods into two categories. Engines look for tasks (engine driven scheduling) or tasks look for engines (task driven scheduling). The first choice has been adopted by the scheduler used in Andorra-I [7]. This model consists of a top scheduler and two sub-schedulers each responsible for AND-parallel and OR-parallel execution respectively. The schedulers partition the engines into flexible teams to distribute tasks effectively. However its estimation of the load of a task is crude because it depends only on its complexity. Complex tasks do not always generate many parallel sub-tasks. Engine-driven scheduling can not always relate efficiently the actual task load with the composition of each team imposing run-time task and engine migration overheads. A unified top scheduler for all forms of parallelism could improve performance further by reducing certain synchronization overheads and the complexity of interfacing between the scheduler and the engines. While the bounded depth-first distribution strategy used by Andorra-I may relate closely to the actual Prolog selection strategy it does not always distribute tasks effectively to engines. Communication costs are not properly quantified because this scheduler was designed for shared-memory multiprocessors with fast communication. If the same approach was used by PAN's scheduler it might often result in a slow-down. A similar engine-driven approach is used by &-Prolog [10].

Engine-driven approaches are also used by MUSE [1], Aurora [12], and the Bristol Scheduler [3]. These systems mainly address the problem of efficient scheduling of OR-parallel tasks controlling speculative OR-work. A drawback of such engine-driven schedulers with this approach is that an idle worker must make a global search for new work, which is a major and time consuming task switch and imposes real overheads. Most task switches involve a global search for new work which significantly affects performance. In MUSE when a worker is idle, its next piece of work will be taken from the deepest (i.e. youngest) node on the richest branch. The measure of richness used is the number of unshared (or private) alternatives on the branch. A disadvantage is that the shared (or public) region of the tree is much larger and the overall computation is slower since backtracking over public nodes is more expensive than backtracking over private nodes despite the

reduction to the number of major task switches. Such scheduling strategies share common characteristics that may affect performance. A considerable number of parallel tasks are identified at run-time, while PAN’s approach detects parallelism mainly at compile time. They are designed for platforms with low communication costs and may fail to quantify much of the run-time overheads of distributed heterogeneous platforms properly that degrade performance. The task switches and the search for new work depend proportionally on the communication overheads among engines.

The significant costs of inter-engine communications in distributed platforms have been considered in the scheduling techniques used in OPERA [4] and PloSys [13] which exploit OR-parallelism. The scheduling is performed by a hierarchy of specialized schedulers operating in parallel to the workers using an approximate representation of the state of the system, while the multisequential computational model of OPERA does not create more parallelism than required by the available resources. To improve its performance OPERA significantly re-engineers the WAM code to implement the scheduling algorithms efficiently. This represents a departure from the use of mainstream Prolog technology on process-based distributed platforms and makes the approach liable to being marginalised as mainstream technology evolves. Similar multi-sequential models have also been adopted by more recent distributed systems like PDP [2] which also implements an extension of the WAM. The current implementation of PDP only uses one scheduler to support the number of available engines. This results in a centralized scheduling scheme that may create bottleneck situations in distributed platforms like PAN with slow communication. The configuration of the team of engines lacks flexibility and does not change dynamically during program execution to adapt to any changes in the distributed platform.

3 Design Choices in PAN

The current implementation of PAN exploits OR-parallelism, independent AND-parallelism and combinations [19]. A suitable granularity control mechanism [21] has been incorporated in PAN, while a second mechanism estimates at compile-time the relative difficulty of a task. The *difficulty* of a task in this case does not only depend on its complexity [8], but also on the number of parallel sub-tasks it may generate. Complex tasks may not generate many parallel sub-tasks and vice versa. The scheduling approach of this research is different in many ways from all methods mentioned. It adopts a dynamic task-driven strategy by trying to reduce the run-time overheads of making distribution decisions that depend on the communication costs while the engine re-allocation and task redistribution schemes keep engines busy at run-time. Task-driven scheduling is dictated by the fact that compile-time analysis has already detected parallel tasks. This considerably reduces the overheads of searching for parallel tasks at run-time and complies with the design choices of platforms like PAN.

Scheduling analysis in PAN uses a *Farmer-Worker* model for engine distribution which generates goal driven ”master-slaves” relations among engines and relates

closely to SLD resolution. As the goals change, these relations should adjust to the program requirements dynamically. A dynamic hierarchy of goals and sub-goals is generated which corresponds to a hierarchy of farmers and workers. Farmers do not interfere with the workers (and their tasks) of other farmers at the same or different level of the hierarchy. The basic mechanism of the proposed scheduling algorithm is *best-first* task distribution in the sense that the more difficult tasks get to use more engines first. The *farmer-worker* hierarchy is particularly suitable for distributed systems in contrast to techniques presented in section 2. It does not have a central scheduling unit but each node in the hierarchy corresponds to a distributed component. Each such component consists of a distributed scheduler, its workers and a local engine pool. Distributed components communicate infrequently, they perform only the necessary communication (detect failure for instance). OR-parallelism in PAN is explored in an AND-parallel manner. The algorithm presented in [14] has been modified to operate effectively under PAN. Therefore PAN does not require separate schedulers to distribute tasks to engines, while the proper management of speculative OR-work is handled by the compile-time analyzer [19] reducing certain run-time overheads (like task switches) imposed by models like MUSE and Aurora.

3.1 Task Distribution and Engine Re-Allocation

When the program is actually executed there is an initial distribution (determined at compile-time) of tasks to engines which corresponds to an initial hierarchy (configuration) of farmers and workers. Difficult tasks get more engines. The reader is referred to [20] for further discussion. However, it remains possible for an engine to process more than one parallel tasks while the number of engines is less than the number of parallel tasks. In this case tasks have to be re-distributed and engines re-allocated. The run-time scheduler for farmers and workers is presented in the abstract algorithms of figures 1 and 2. Engine allocation is dynamic in the sense that each farmer may have a different number of workers during program execution or when the program and the initial goal change to adjust to the distribution of tasks. The implementation can determine a maximum number of workers a farmer can have to avoid bottleneck situations. Experimental results suggest that a farmer should not have more than 4 workers in PAN when running the programs presented in section 5.

A farmer can be viewed as a worker for a farmer at a higher level. A farmer communicates only with its workers by sending engines and tasks (figure 1, lines 9) and receiving results (figure 1 lines 7 and 8) from them or with its parent-farmer (figure 1 line 1, 5 and 6). If the farmer detects failure, (figure 1 lines 4, 6 and 8) it initiates the abort and failure mechanism (presented in following sections). The farmers **sort** any requests for more engines from their workers (figure 1 line 9) in a best-first manner re-distributing any available engines (figure 1 line 10). The (flexible) order in which the farmer processes tasks locally (figure 1 line 2) is also determined by the implementation of the algorithm.

<i>farmer(Tasks, Workers, Parent_Farmer, Pool, Waiting_List, Result)</i>	
if number_of(Tasks)-1 > number_of(Workers) then	1
ask(Parent_Farmer, number_of(Tasks)-number_of(Workers)-1)	
process_locally(task, ResultA) task∈Tasks	2
if ResultA≠fail then	3
Result ← Result∪{ResultA}	
Tasks ← Tasks - {task}	
else	4
failure_mechanism	
if message_from(Parent_Farmer)=Engines then	5
Pool ← Pool∪Engines	
Workers ← Workers∪Engines	
if message_from(Parent_Farmer)=abort then abort_mechanism	6
if message_from(Worker)=(idle, ResultB, Engines) then	7
Result ← Result∪ResultB	
Pool ← Pool∪{Worker}∪Engines	
if message_from(Worker)=fail then failure_mechanism	8
if message_from(Worker)=request then	9
Temp_List ← Waiting_List∪{Worker}	
sort(Temp_List, Sorted_List)	
wait_until Pool≠{}	10
repeat	
distribute(Pool, Sorted_List)	
until Sorted_List={ } OR Pool={ }	
Waiting_List ← Sorted_List	
if number_of(Tasks)=0 then	11
wait_until state_of(Workers)=idle	
send_to(idle, Result, Workers), Parent_Farmer)	
if number_of(Tasks)>1 then	12
if state_of(Workers)=idle then	13
repeat	
send_to(task, engine) task∈Tasks, engine∈Pool	
Tasks ← Tasks - {task}	
Pool ← Pool - {engine}	
until number_of(Tasks)=1 OR Pool={ }	
farmer(Tasks, Workers, Parent-Farmer, Pool, Waiting_List, Result)	14

Fig. 1 *Farmer* Execution Protocol

Best-first task distribution generates a *distributed hierarchy of parallel tasks* as well containing sorted potential parallel tasks. Local task processing and message handling is performed in a concurrent manner (using asynchronous communication among engines) to limit any response delays. Workers communicate only with their farmer to receive tasks and engines (figure 2 lines 6, 7), return the results (figure 2 line 9) and send time-stamped requests to their farmer (figure 2 line 1). If a worker receives an engine from the farmer it becomes a farmer itself (figure 2 line 7), otherwise it processes all tasks locally and stays in worker mode (figure 2 line 2). If a worker detects failure, it initiates the abort & failure mechanism (figure 2 lines 3, 5). Best-first scheduling can be slightly relaxed and transformed to breadth-first distribution in order to keep idle engines busy and improve performance. The **distribute** function presented in figure 2 allocates tasks to the engines in a descending order of the processing capabilities of the engines residing in the pool. As a result the work load tends to be more balanced making the proposed algorithm more attractive.

```

worker(Tasks, Farmer, Results)

if number_of(Tasks) > 1 then ask(Farmer, number_of(Tasks)-1)           1
process_locally(task, ResultA) task ∈ Tasks                             2
if ResultA ≠ fail then                                                 3
    Result ← Result ∪ {ResultA}
    Tasks ← Tasks - {task}
else                                                                     4
    send_to(fail, Farmer)
    failure_mechanism
if message_from(Farmer)=abort then abort_mechanism                     5
if message_from(Farmer)=NewTasks then                                  6
    worker(NewTasks, Farmer, NewResult)
if message_from(Farmer)=Engines then                                   7
    farmer(Tasks, Engines, Farmer, Engines, {}, ResultB)
    Result ← Result ∪ ResultB
if Tasks ≠ {} then worker(Tasks, Farmer, Results)                       8
if Tasks = {} then send_to((idle, Result, {}), Farmer)                 9

distribute(Pool, Waiting-List)

W ← head(Waiting_List), Engines ⊆ Pool
send_to(W, Engines)
Waiting-List ← tail(Waiting_List)
Pool ← Pool - Engines

```

Fig. 2 *Worker* Execution Protocol

3.2 Abort & Failure Mechanisms

To make the system *faster* and *closer to standard* Prolog execution the farmer-worker protocol has been given an *Abort Process* mechanism. This mechanism can be effective at controlling speculative parallelism.

1. All workers often check for any messages from their farmer.
2. When a farmer receives a *fail* message from a worker it sends an *abort* message to other attached workers and stops local task processing.
3. When a worker receives an *abort* message, it aborts any processing, notifies the farmer that it has *aborted* all processes and returns to the engine pool.
4. The farmer waits until it receives the *aborted* message from all its workers.
5. Then the farmer (and any associated goal) fails.

The procedure is applied recursively to sub-farmers. An *abort* message is distributed to engines in a top-down manner but engines return the *aborted* message and return to the pool in a bottom-up manner. We could speed up the whole aborting mechanism if we let a sub-farmer send a *fail* message to its farmer earlier than the actual *aborted* message. As a result the workers would abort almost in parallel with the other workers. The farmer aborts (and then fails) only if it receives the *aborted* message from every attached sub-farmer and worker to guarantee *safe process abortion* in the sense that all workers actually stay idle in the

pool having stopped all computations and communication. The proposed mechanism is particularly suitable for distributed heterogeneous platforms because it quantifies effectively the communication costs and initiates useful operations to be performed by the engines in a concurrent manner while time consuming inter-engine communication takes place.

4 Comparisons

This *farmer-worker* model of dynamic best-first engine re-allocation and task re-distribution is a new scheduling scheme for distributed execution of Prolog. It aims to improve performance by reducing the complexity of interfacing and synchronizing among the scheduler and a large number of engines, and keep communication overheads low. Its philosophy is different to allow it to perform better on distributed platforms like PAN in comparison to models like [7, 1, 12, 3, 4, 2] which are designed for shared-memory multiprocessors. It is task-driven and distribution-oriented which conforms with the design choices of process-based heterogeneous platforms. There are several distributed and de-centralized scheduling components to make the model more scalable in contrast to the PDP and reduce inter-engine communication. The farmers spend time scheduling for a small number of workers, therefore the efficiency is improved and bottleneck situations are minimized, while reasonable control of task and engine migration is achieved at little extra cost. The maintenance of a distributed pool does not consume much of the engine resources and the engines in the pool are quickly made available to act as workers. The run-time engine distribution is dynamic and may easily adjust to different kinds of tasks which make it more flexible than other scheduling models that partition engines into fixed numbers. The hierarchy changes dynamically, workers become farmers on demand to process parallel tasks better. The scheme provides fair engine distribution. A "best-first" scheduling policy has the ability to process difficult tasks using suitable system resources providing a good degree of the work load balancing. Other scheduling policies may not always guarantee fair distribution of the engines. The hierarchy of parallel tasks reduces task switches while preserving (to some extent) the usual Prolog execution strategy and includes some of the attractive characteristics of MUSE at little extra cost while the engines are informed of possible failure effectively. The scheduling control is done at the Prolog level and its implementation does not re-engineer the WAM (in contrast to PDP and OPERA) which complies with the design choices of PAN and adds flexibility as mainstream technology evolves and improves the system's portability and maintainability.

The model imposes some overheads which relate mainly to the frequency of the communication between a farmer and its workers and depend mainly on the characteristics of the platform. The actual frequency is left to an implementation. PAN follows PMS-Prolog [18] reasons in not supporting backtracking in communication. This scheduling scheme does not support distributed backtracking either in contrast to platforms like Delta-Prolog [6] and CS-Prolog [9].

5 Performance

PAN provides a suitable platform to determine if the proposed controls adapt well to the changing needs of a heterogeneous multiprocessor. Large input sizes have been used to provide long running non-trivial problems. All programs are listed in [20]. Direct comparison of PAN with parallel Prologs on shared-memory multiprocessors are not always reasonable, they usually perform better than distributed platforms as argued in [11] and [5]. It isn't always feasible to compare the performance of distributed platforms either because they have different hardware configurations making it difficult to establish a general and fair comparison metric. The benchmarks were run under PAN using SICStus Prolog 3.5 and PVM 3.3.11 on a variety of SUN, DEC and IRIX Unix workstations. The numbers in tables represent the relative performance improvement RPI due to parallel execution in comparison to sequential execution. RPI is calculated using the formula $RPI = \frac{SE}{PE}$ where SE is the sequential execution time (in seconds) and PE the parallel execution time (in seconds). The average number of the best three runs is used. To provide a fair comparison metric of SE for this heterogeneous platform each goal is run on all engines participating in a given PAN session. The average of the best three runs is chosen to represent the execution time SE_i of an engine i . Then the overall sequential execution SE is calculated as the average value of all SE_i .

5.1 AND-parallel Execution

To illustrate the performance of the model for AND-parallelism the QuickSort program, the MergeSort program, the Big Integer Matrix Multiplication program, and the Perfect Numbers program were run under PAN. PAN can process tasks at a certain rate due to the large communication costs. If a program produces AND-tasks faster than PAN can consume them then the proposed model incurs extra run-time overheads because it schedules potential parallel tasks which are processed locally instead. The Matrix Multiplication program generates four medium-grained parallel tasks on each recursion, but the rate of generation is not reasonably close to the rate PAN can process them. Such medium-grained programs perform better on shared-memory multiprocessors. In contrast, the rate that the QuickSort and MergeSort programs generate coarse-grained parallel tasks is reasonably close to the rate that PAN can effectively process them. The Perfect Numbers provide the best improvement, indicating that for non-trivial and coarse-grained applications this model distributes tasks effectively to engines while controlling the communication overheads and exploiting good degrees of parallelism.

QSort	List Input Size			
	750	1000	2000	3000
4 Eng.	2.109	2.180	3.087	3.677
8 Eng.	2.712	3.269	4.240	4.498
12 Eng.	3.067	3.661	4.867	5.218
16 Eng.	3.643	4.443	5.652	5.877

MSort	List Input Size			
	750	1000	2000	3000
4 Eng.	1.573	1.843	2.233	2.594
8 Eng.	2.087	2.214	2.611	3.117
12 Eng.	2.531	3.072	4.107	4.565
16 Eng.	2.707	3.812	4.487	5.066

Matrix	NxN Matrix Input Size		
	30	45	60
4 Eng.	1.663	1.629	1.657
8 Eng.	1.720	1.678	1.740
12 Eng.	1.859	1.816	1.941
16 Eng.	2.146	2.122	2.414

Perfects	Integer Input Size		
	100	300	500
4 Eng.	2.159	3.045	3.828
8 Eng.	4.849	6.372	7.090
12 Eng.	6.150	9.220	10.195
16 Eng.	7.050	11.562	12.887

It is encouraging to see that performance improves as the size of tasks and the number of engines increases indicating that the scheduling scheme can partition effectively the work load and can also adapt to the changing configuration of the platform. &-Prolog provides a speed-up of 4.9 for QuickSort(1000) running on 10 nodes of a shared-memory multiprocessor. The AND-OR-parallel distributed executor [17] improves the performance of QuickSort(2000) by 2.7 on 30 processors and PDP improves the performance of QuickSort(700) 2.9 times running on 15 processors. &-Prolog provides a linear speed up of 10 for Matrix(50) running on 10 processors, but distributed platforms like PDP provide a speed up of 1.85 for Matrix(75) on 15 processors. Finally PDP provides a speed up of 2.6 for MergeSort(500) on 12 processors.

5.2 OR-parallel Execution

Analysis gets more complicated when it comes to OR-parallel execution. Programs like Permutations or naive N-Queens that generate fine-grained parallelism are not expected to perform that well under platforms with considerable communication costs as argued in [11,5]. Preliminary results showed that PAN is not an exception. Alternative benchmarks can be used to illustrate the performance of distributed platforms. The OR-Tree and Deep Fail programs are variations of benchmarks used in the performance analysis of distributed systems in [11]. When OR-tasks fail, the OR-interpreter switches to other unexplored branches of the execution tree. These failure-driven task switches are non-trivial and time consuming operations. However the tables indicate that PAN copes adequately.

OR-Tree	Integer Input Size			
	1000	3000	5000	7000
4 Eng.	1.374	2.737	3.220	3.907
8 Eng.	3.587	5.852	6.869	7.816
12 Eng.	6.084	9.057	10.290	10.968
16 Eng.	6.522	9.800	12.689	14.030

Deep Fail	Integer Input Size			
	1000	3000	5000	7000
4 Eng.	1.843	2.631	3.066	3.775
8 Eng.	2.375	3.677	4.665	5.503
12 Eng.	2.998	5.257	6.096	7.120
16 Eng.	3.851	6.361	8.288	9.030

5.3 AND-OR-parallel Execution

Implementation schemes combining AND and OR parallelism typically pay a penalty in the form of a higher control overhead. The following table contains the performance numbers of the *synthetic* benchmarks used in the performance analysis of PDP. They generate AND-under-OR and OR-under-AND parallelism respectively. PAN performs better exploiting the AND-under-OR parallelism of

the `synthetic-1` benchmark because it requires fewer task switches between the OR-interpreter and the Prolog engine in comparison to OR-under-AND parallelism generated by the `synthetic-2`. This indicates that PAN favours the use of the OR-interpreter at the top levels of the execution tree while the tasks in lower levels can be processed either sequentially or in AND-parallel.

AND-OR-parallelism	<i>Synthetic-1</i>	<i>Synthetic-2</i>
4 Eng.	2.537	2.032
8 Eng.	3.583	3.079
12 Eng.	4.302	3.705
16 Eng.	4.423	3.819

For the `synthetic-1` benchmark PDP provides a speed up of up to 4.5 and for the `synthetic-2` benchmark a speed up of up to 4.6. The latter benchmark performs better when run under the PDP system because OR-parallel execution is realized by extending the WAM which imposes no task switches between the Prolog engine and the OR-mechanism which is the case in PAN. But PAN also performs reasonably well using mainstream Prolog technology.

6 Summary - Future Work

A scheme for scheduling the execution of parallel tasks on a process-based heterogeneous distributed multiprocessor has been presented. The model uses distributed scheduling components that communicate infrequently and adopts a dynamic task-driven approach to adapt to the changing nature of such platforms. Each component contains a farmer engine responsible for the distribution of parallel tasks to a number of workers and the maintenance of a distributed pool. An abort & failure distributed mechanism has also been incorporated into the model to reduce the speculative work. Preliminary results indicate that the proposed techniques can facilitate the execution of distributed tasks efficiently and improve the performance of programs. These figures indicate that PAN performs better running time consuming and non-trivial applications rather than fine-grained parallel tasks. Further research will focus on experimenting with a wider range of applications to improve the design and the implementation of the proposed scheduling model and determine a wider of applications that can benefit from a distributed platform like PAN.

References

1. K.A.M Ali and R. Karlsson. Scheduling OR-parallelism in MUSE. In K. Furukawa, editor, *8th International Conference on Logic Programming*, pages 807–821, Paris, June 1991.
2. L. Araujo and J.J. Ruz. A parallel Prolog system for distributed memory. *International Journal of Logic Programming*, 33(1):49–79, October 1997.
3. A. Beaumont, S.M. Raman, and P. Szeredi. Flexible scheduling of OR-parallelism in Aurora: The Bristol scheduler. Technical report, Department of Computer Science, University of Bristol, October 1990.

4. J. Briat, M. Favre, C. Geyer, and J.C. de Kergommeaux. OPERA: Or-parallel Prolog system on Supernode. In P. Kacsuk and M.J. Wise, editors, *Implementations of Distributed Prolog*, pages 45–64, John Wiley, Chichester, 1992.
5. G.F. Coulouris and J. Dollimore. *Distributed Systems: concepts and design, 2nd edition*. Addison Wesley, 1994.
6. J.C. Cunha, P.D. Medeiros, M.B. Carvalhosa, and L.M. Pereira. Delta-Prolog: A distributed logic programming language and its implementation on distributed memory multiprocessors. In P. Kacsuk and M.J. Wise, editors, *Implementations of Distributed Prolog*, pages 335–356, John Wiley, Chichester, 1992.
7. I. de Castro Dutra. A flexible scheduler in the Andorra-I system. In Anthony Beaumont and Gopal Gupta, editors, *Proceedings of ICLP '91- Pre-Conference workshop on Parallel Execution of Logic Programs*, pages 70–82, Paris, June 1991.
8. S.K. Debray, P.L. Garcia, M.V. Hermenegildo, and N.W. Lin. Estimating the computational cost of logic programs. In B.L. Charlier, editor, *Static Analysis Symposium 1994*, pages 255–265, Namur, Belgium, Sept 1994.
9. I. Futo. The real time extension of CS-Prolog professional. In J. Barklund, B. Jayaraman, and J. Tanaka, editors, *ICLP'94 - Workshop on Parallel and Data Parallel Execution of Logic Programs*, Santa Margherita Ligure, June 1994.
10. M.V. Hermenegildo and K.J. Greene. The &-Prolog system: Exploiting independent AND-parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.
11. P. Kacsuk and M.J. Wise. *Implementations of Distributed Prolog*. John Wiley, Chichester, 1992.
12. E. Lusk, D.H.D. Warren, and S. Haridi. The Aurora OR-parallel system. *New Generation Computing*, 7(2,3):243–271, 1990.
13. E. Morel, J. Briat, J.C. de Kergommeaux, and C. Geyer. Side-effects in PloSys Or-parallel Prolog on distributed memory machines. In M.J. Maher, editor, *ICSLP'96- Compulog Net Meeting on Parallelism and Implementation Issues*, Bonn, September 1996.
14. E. Shapiro. An OR-parallel algorithm for Prolog and its FCP implementation. In J.L. Lassez, editor, *Proceedings of Forth International Conference on Logic Programming*, pages 311–337, Melbourne, May 1987.
15. H. Taylor. Assembling a resolution multiprocessor from interface, programming and distributed processing components. *Computer Languages*, 22(2,3):181–192, 1996.
16. J. Turek, K. Pattipati, P. S. Yu, and J. Wolf. Scheduling parallelizable tasks: Putting it all on the shelf. In *ACM SIGMETRICS and PERFORMANCE International Conference on Measurement and Modeling of Computer Systems*, pages 225–236, Rhode Island, USA, June 1992.
17. A. Verden and H. Glaser. An AND-OR-parallel distributed Prolog executor. In P. Kacsuk and M.J. Wise, editors, *Implementations of Distributed Prolog*, pages 143–157, John Wiley, Chichester, 1992.
18. M.J. Wise. Experience with PMS-Prolog: A distributed, coarse-grain-parallel Prolog with processes, modules and streams. *Software Practice and Experience*, 22(2):151–175, 1993.
19. G. Xirogiannis. Compile-time analysis of freeness and side-effects for distributed execution of Prolog programs. In T. Sellis and G. Pagkalos, editors, *6th Hellenic Conference on Informatics*, pages 701–722, Athens, December 1997.
20. G. Xirogiannis. *Execution of Prolog by Transformations on Distributed Memory Multi-Processors*. PhD thesis, Dep. of Computing & Elec. Engineering, Heriot-Watt University, Edinburgh, Scotland, 1998.
21. G. Xirogiannis. Granularity Control for Distributed Execution of Logic Programs. In *18th International Conference on Distributed Computing Systems* to appear, Amsterdam, May 1998.