A Flexible Scheduler for the Andorra-I System*

Inês de Castro Dutra†
Department of Computer Science
University of Bristol
Bristol BS8 1TR
e-mail: ines@compsci.bristol.ac.uk

Abstract

A flexible scheduler for distributing and- and or- parallel work in the Andorra-I system is presented. The scheduler allows workers to move freely between teams. Its strategy is based on estimates of and- and or- parallelism available in the execution tree. Preliminary results show that in most cases we can reach or surpass the best performance of Andorra-I running with any fixed team configuration. Since the teams are configured dynamically and automatically, the user does not need to be concerned with choosing an appropriate configuration. Further work is being done to improve the scheduler by using data obtained from compile-time analysis.
Keywords: And/Or-parallelism, Multiprocessors, Scheduling.

1 Introduction

There are two main sources of parallelism in logic programming, namely or- and and-parallelism. Or- parallelism is exploited when several alternative clauses for a goal are executed in parallel. And- parallelism is exploited when we execute two or more goals of the same clause simultaneously. Exploitation of full or- and and- parallelism is limited by the number of physical processors available in a system. And- parallelism is also limited by the interdependence among goals in a clause.

One very interesting way of exploiting and- parallelism in logic programs is used in Andorra-I [18, 19], where **determinate** goals are executed first and concurrently, according to the *Andorra Principle* [26]. Determinate goals are the ones that match at most one clause in a program. This way Andorra-I exploits dependent and- and independent determinate and- parallelism. Eager execution of determinate goals also results in a reduced search space, because unnecessary choicepoints are eliminated.

The Andorra-I parallel logic programming system exploits not only dependent and-parallelism, but also or- parallelism as in Aurora [16]. The system is composed of an engine, responsible for the execution of the Andorra-I programs [28, 29, 20], and a scheduler, whose main function is to find new tasks for the engine. A processing element that performs computation in the system is called a *worker*. Workers are divided into *teams* and cooperate with each other in order to share available and- work. Each team has a

*Research supported by CNPq, Brasilian Research Foundation under grant 202270/89.0

master, which is responsible for creating new choicepoints and *slaves* that are managed by their master. Teams of workers cooperate to share or- work.

The earlier status of the Andorra-I system was such that the organisation of teams could not be changed dynamically. The teams were of fixed size, the workers could not move from one team to another, and only master workers could grab or- work. This organisation of the system, although acceptable for some applications, is too rigid for a large number of applications where the degree of and- and or- work varies along the course of the execution. In such applications, the available parallelism is not completely exploited, if we have fixed the configuration of teams a priori.

Motivated by this limitation, we have designed a flexible scheduler and have implemented a simple version. The results obtained are shown in section 2.2. Although these results are better than those produced by Andorra-I with a fixed team configuration, we feel that the top-scheduler has some limitations, as discussed in section 3. This in turn led us to investigate other more intelligent strategies based on information provided by global analysis.

In the next section we discuss the top-scheduler strategy and show results obtained so far. In section 3 we discuss some issues and ways of enhancing the current strategy. In the last section we discuss alternatives of implementing more intelligent algorithms which will be investigated in the future.

2 The Top-Scheduler

The current design of the flexible scheduler takes account of the fact that there already exist distinct schedulers to deal with and- and or- work. The novel part of the flexible scheduler, therefore consists of a *top-scheduler* which is responsible for making decisions when a worker does not find either and- or or- work. The or-scheduler is responsible for finding or- work, i.e. an unexplored alternative in the tree. Any Aurora or-scheduler [2, 21, **?**, 3, 8] can be plugged into the system through the Aurora scheduler interface [7, 22, 23]. The and-scheduler is responsible for finding and- work, which corresponds to a goal in the run queue (list of goals not yet executed) of a worker in the same team.

Earlier schedulers were designed based on software engineering principles of having a clear and distinct interface between engine and scheduler. The top-scheduler was also designed via its interface functions. As a first step, the interactions between the top-, and-, and or- schedulers and the engine were identified and formed the basis of the interface functions. Details about the top-scheduler interface and its implementation can be found in [10].

In the subsequent sections we discuss the top-scheduler strategy and show the results obtained so far.

2.1 The Top-scheduler Strategy

In Andorra-I, as described in the previous section, a worker can be a master or a slave of some team. In a flexible scheme we allow migration of workers from one team to another. A master can move to another team and become a slave when it runs out of or- work. A slave can move from its team to another whenever no and- work is found inside its team. It can also become a master of a new team.

The main goals of the top-scheduler are (a) to decide when an idle worker will change its type (master/slave) and (b) to choose the best team for a slave, while considering the cost of exchanging workers between teams. In order to avoid workers exchanging teams very often, we maintain a counter that registers the number of times a worker calls the top-scheduler without changing teams. As regards (a), the decision to change the type of a worker is based on a relation (*greatness factor*) between the current amount of and- and or- work available in the tree. Since we do not have sufficient information about the degree of parallelism in the application, this relation is tuned to adjust to the amount of parallelism available in the application. In fact, it represents the weight between and- and or- work. The best team is the one with more amount of and- work per worker (the one that has greatest *load*).

```
MASTER CALLS TOP-SCHEDULER:

Begin
  calculate the amount of or-work;
  calculate the amount of and-work;
  ratio = and-work/or-work;
  If ( ratio > greatness factor )
      master becomes slave;
      calculate the load of teams;
      move new slave to
         the best team;
End.
```

```
SLAVE CALLS TOP-SCHEDULER:

Begin
  calculate the load of the teams;
  If ( there is a team with
       greatest load )
      move worker to the
         best team;
  Else
      calculate the amount of or-work;
      calculate the amount of and-work;
      ratio = or-work/and-work;
      If ( ratio > greatness factor )
          slave becomes master;
          new team is created;
End.
```

Figure 1: Top-scheduler strategy

The basic algorithm that implements the strategy is shown in Figure 1. The amount of or-work is provided by the or-scheduler and corresponds to the number of live nodes (choicepoints that still have alternative clauses to be expanded) in the or-branch of a master worker. The amount of and-work is provided by the and-scheduler and corresponds to the total number of goals in a run queue. The load of a team is the quotient $\frac{run\ queue\ size}{team\ size}$. The *greatness* factor, as mentioned before, is varied according to the degree of and- and or- parallelism available in the application.

The worker cycle with respect to finding work is shown in Figure 2. The top-scheduler can be called from the or-scheduler when no or- work is found for a master worker. In this case, the top-scheduler has two alternatives: (a) let the master worker continue to look for or- work; or (b) make the master a slave and move it to the *best* current team. The top-scheduler can also be called by the and-scheduler if no local and- work (in the same team) is found for a slave worker. In this case, it has three alternatives: (a) let the slave worker continue to look for local and- work; (b) move this slave from its current team to another better team (the *best* currently available); (c) make the slave a master of a new

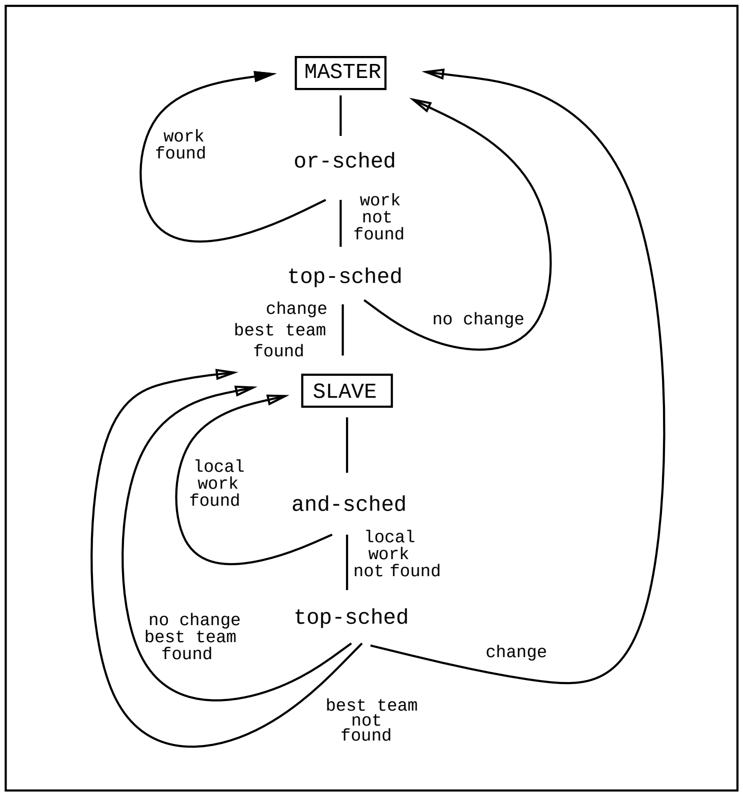

Figure 2: Life cycle of a worker

team.

2.2 Preliminary Results

In Andorra-I with fixed teams, the team configuration must be specified from the beginning. The system can behave badly if, because of lack of knowledge about the quantity of parallelism available in the application, the user cannot configure the right team. Moreover, for irregular applications where the type of parallelism varies during execution, the system can still behave badly whatever initial configuration is selected. When designing the top-scheduler, we thus expected to reach the best speedups achieved by Andorra-I for programs with either and- parallelism or or-parallelism, and better or comparable results for examples that mix both kinds of parallelism.

Table 1 shows the speedups obtained by Andorra-I when using fixed teams and the flexible scheduler. The data related to the flexible scheduler shown in table 1 are the results obtained by giving priority to and-work with a *greatness* factor of 3. This factor was chosen as a starting point to measure the relation between and- and or- work as we do not yet know what is an optimal relation. Best and Worst speedups were obtained from 10 runs of the programs. For the version with fixed teams the Best and Worst speedups were taken, respectively, from the best and worst times obtained among all possible combination of workers. So, for example, `cross6`, had its best speedup running with 2 masters and 2 slaves, whilst it had its worst speedup running with 4 masters. All the examples run with the flexible scheduler started from one team of 4 workers.

Class	Pgm	Sp.fix.teams		Aver.	config	Sp.flex.sch.		Aver.
		Best	Worst		best sp.	Best	Worst	
artif. pgms.	and	3.04	0.99	2.015	1M 3S	3.08	2.64	2.860
	or	3.60	1.04	2.320	4M	3.76	3.53	3.645
	and/or	2.02	1.16	1.590	3M 1S	2.21	2.04	2.125
	or/and	3.20	2.33	2.765	3M 1S	3.30	2.49	2.895
	mixed	2.18	1.26	1.720	4M	2.01	1.74	1.875
and	fib	3.29	0.98	2.135	3S 1M	3.26	2.80	3.030
	mergesort	2.87	1.02	1.945	3S 1M	2.85	2.34	2.595
	nrv50	2.14	0.94	1.540	3S 1M	2.28	2.00	2.140
	nrv150	2.94	0.97	1.955	3S 1M	3.20	2.92	3.060
and>or	bqu	1.76	1.00	1.380	3S 1M	1.77	1.30	1.535
	scanner	1.92	1.25	1.585	3S 1M	1.88	1.51	1.695
and=or	cross6	1.93	1.38	1.655	2S 2M	2.03	1.56	1.795
	sets	2.99	2.11	2.550	2S 2M	2.94	2.39	2.665
or>and	db_conery	1.33	0.80	1.065	1S 3M	1.67	1.17	1.420
	qu_vitor	3.73	1.33	2.530	4M	3.88	3.80	3.840
	qu_reem	3.21	1.78	2.495	4M	3.50	3.28	3.390
	money	3.65	1.23	2.440	4M	3.75	3.70	3.725
	houses	3.31	1.54	2.425	4M	3.34	3.04	3.190
	qu_evan	3.81	0.87	1.840	4M	3.75	3.75	3.750
	map	3.35	0.95	2.150	4M	2.63	2.54	2.585
	rem_par	3.16	1.02	2.090	4M	2.66	2.45	2.555
or>>and	mutest	3.20	0.93	2.065	4M	3.12	2.89	3.005
	plan	2.70	1.15	1.925	4M	2.23	1.83	2.030

Table 1 - Speedups - FT x NFT (4 workers)

The first column of Table 1 represents the class of the applications. The first group (artificial programs) consists of well behaved artificial examples used to study the strategy of the top-scheduler. From this group program `and` only has and- parallelism and does not create any choicepoints, program `or` has no and- parallelism and creates a single choicepoint in the beginning. Program `and/or` has two phases. The first one has only and-parallelism and the second one creates a choicepoint. Program `or/and` has an or- parallel phase followed by an and- parallel phase. Finally, program `mixed` combines everything. The other programs are more realistic and are classified according to the degree of and- and or- parallelism available. The columns named "Aver." represent the average speedups and show that overall the results produced by Andorra-I under the top-scheduler strategy are much better than those obtained with a fixed configuration of teams.

We chose to run the examples, at first, using 4 workers, but each example has a least number of processors with which it should reach the theoretical or approximate expected speedup. If we consider an infinite number of processors, it is not difficult to find the best theoretical speedup. But if we limit the number of processors, it is yet not clear (a) what is the least number of processors we should use to obtain the maximum speedup; and (b) if we have a number of processors below the least number, what would be the best speedup for that number of processors. For the small examples of table 1, it is relatively easy to find the expected speedups for programs `and` and `or` running with 4 workers (which is approximately 4.0), but for complicated shapes of trees, this is a hard task.

For program `and`, as expected, Andorra-I running under the top-scheduler strategy produces speedups similar to the ones obtained by running Andorra-I with a fixed team of 1 master and 3 slaves. For program `or`, the result shown was also expected, because the speedup using the flexible scheduler should be comparable to Andorra-I running with a fixed team of 4 masters. We conclude that there is not much initial overhead of arranging the workers in 4 teams, given that we started with just one team of 4 workers.

For the mixed examples, Andorra-I running under the top-scheduler strategy behaved better or comparable to the version with fixed configuration of teams. Interestingly, program `mixed` shows slightly better speedup when using fixed teams. We observe similar results for some of the "real" examples. As discussed later, instant snapshots of the system may mislead the top-scheduler and sometimes it may make wrong decisions.

For the more realistic examples with just and- parallelism, Andorra-I with flexible scheduler behaves akin to Andorra-I with fixed teams, as expected.

In contrast, for the examples that have a balanced combination of and- and or- work, the top-scheduler performance depends strongly on its luck about deciding to change workers from one team to another. As an example, the program `sets` produced better speedups when the top-scheduler gave priority to or-work and the greatness factor was 0 (which means that the workers were allowed to change their types for any ratio between and- and or- work).

For the examples with more or- than and- parallelism, we can observe that as the amount of or- work in the program increases (following down the table), Andorra-I under the top-scheduler produces speedups worse than Andorra-I running with fixed teams. We believe that this is a consequence of the current strategy of giving priority to and-work. More tests are being made to confirm this.

In most cases, we can reach the best speedups for Andorra-I with fixed team size and, as expected, we can get better performance for the examples that have mixed parallelism and that can benefit from the current top-scheduler strategy. Obviously the worst performance using non fixed teams is always much better than the worst performance using fixed teams, because we can always find a fixed team configuration that makes the system behave badly. Additionally, from the point of view of the user it is much nicer to have a system where the programmer does not need to worry about the number of masters/slaves that should be used to run an application.

Figure 3 compare speedups obtained by varying the number of processors for program `sets`, which has a balanced combination of and- and or- parallelism. In order to obtain the best speedups for fixed teams it was necessary to use different combination of masters and slaves. For the flexible scheduler and 4 workers the result was obtained by running the strategy that gives priority to or-work and uses a greatness factor of 0 (this was the best result obtained after running `sets` and varying the greatness parameter to values 0, 10, 20, 50, 100, 1000). All other speedups were obtained by running the strategy that gives priority to and-work and uses a greatness factor of 3. Other results show that when we

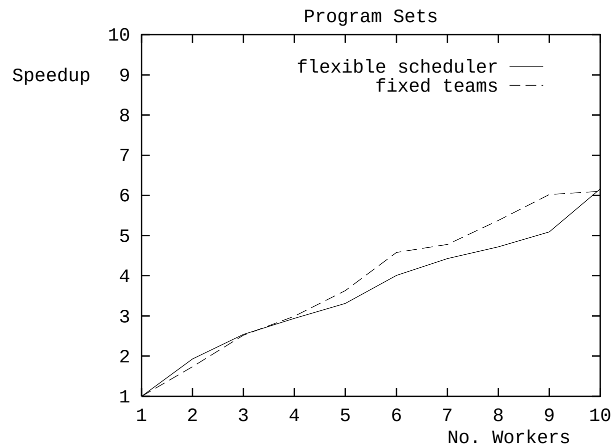

Figure 3: Speedups achieved by running the `sets` program

change the priority of giving work and the greatness factor we can benefit more from the parallelism available in the program, although this is not a regular behaviour according to the data collected so far.

3 Issues in Flexible Scheduling

We believe that the strategy we use to move workers from one team to another is still rather crude, although it produces results comparable to the best obtained using Andorra-I with fixed teams. In some examples, the performance could be better if decisions were not just based on immediate information about the program execution. For instance, two instants of the system configuration for the crossword puzzle problem were taken: time of decision (instant 1) and time of leaving the top-scheduler (instant 2). The following figure appeared:

	and_richness value	or_richness value
instant 1:	0	1
instant 2:	17	2

Based on the values at instant 1, the top-scheduler decided to create a new team because and_richness was equal to zero and there were alternatives left in the tree. But, despite the fact that `cross6` performed better under the flexible scheduler strategy, this decision was not the best, because at instant 2 and_richness was much greater than or_richness. Obviously, this decision could be in fact the right one if we knew beforehand the degree of determinacy of the goals in the run queues and the task sizes below these goals in the tree.

Sometimes the and-richness can mislead the top-scheduler (causing an "illusive" effect), because the run queues can contain many non determinate goals.

In addition, other issues should be taken into account in order to improve the performance of the system. For example, what should be the decision if there is no immediate and-work in the execution tree (which means and-richness equal to zero), and if the amount of or-work is equal to zero? How many processors should be used to achieve the maximum possible speedup without degrading the performance? What if the amount of and- and or- work are the same? How to use measure of the overheads of moving workers from one team to another?

Further work is being done in order to modify the Andorra-I preprocessor [19] and collect data that we think is important to improve the flexible scheduler: size of the paths below the goals in the run queues, type of goals in the run queues (totally determinate, totally non-determinate or with a probability of becoming determinate) and a rough approximation of the amount of parallelism below the goals estimated by $\frac{\textit{total number of nodes}}{\textit{longest path}}$, where *total number of nodes* is the number of nodes necessary to execute the goal sequentially, and *longest path* is the size of the longest branch below the goal by counting determinate reductions that can be done in parallel as only one node.

4 Future Topics

The problems with the current strategy mentioned show that a more intelligent strategy should be adopted to distribute and- and or- work. In order to do that, we assume that it is possible to obtain useful information (like priorities for execution of goals, *speculativeness* in the scope of cuts and commits, task granularity, degree of determinacy, graph of the program, rough amount of and- and or- work) from global analysis. Some work in the literature has shown that this task is feasible [9, 24, 4, 19, 11, 25, 27]. We are therefore going to concentrate on scheduler strategies that can use the mentioned information. We will be considering decentralised dynamic (adaptive) load balancing strategy, since the **exact** shapes of the search trees of logic programs are unpredictable.

Scheduling schemes [17] can have two policies: *supply-driven* or *demand-driven*. A supply-driven policy determines whether to accept a task whenever a new task becomes available. The latter allocates a processing element (worker) to a task, whenever the policy allocation notices that the worker is lightly loaded. A demand-driven policy is simpler and involves less overhead to workers as they are not interrupted by the scheduler. Also a policy can be *pre-emptive* or *non-preemptive*, which means that a worker can suspend or continue some task voluntarily and start another task.

Our approach will be to use a demand-driven and (perhaps) a pre-emptive scheduler which will navigate through the tree looking for *better* tasks. The guiding principle will be based on a rough graph of the computation tree (presumably the theoretical execution tree), the degree of determinacy of the goals, the priority of execution of the goals and the load of each processor. Given the graph of the computation tree, we have information about the longest path, that in a system to find all solutions, is particularly important because it gives us the least possible time to find all solutions in parallel. With information about the degree of determinacy of a goal we can measure the granularity of the tasks below that goal. We assume a pre-emptive policy as we intend to use priorities to schedule work. This way, if the worker is executing some speculative work (because there was no useful piece of work available) and some useful work appears in the tree, the scheduler can suspend the active worker and switch it to the useful piece of work. Our main objective will be to maintain the height close to that of the theoretical tree whenever possible and to avoid frequent task switching.

Example (1):

```
1) f(X,Y) :- p(X), q(Y).         6) p1(X) :- g1(X).
2) f(X,Y) :- p1(X), q1(Y).       7) q1(X) :- g2(X).
3) p(1).                         8) g1(a).
4) p(2).                         9) g2(b).
5) q(2).
```

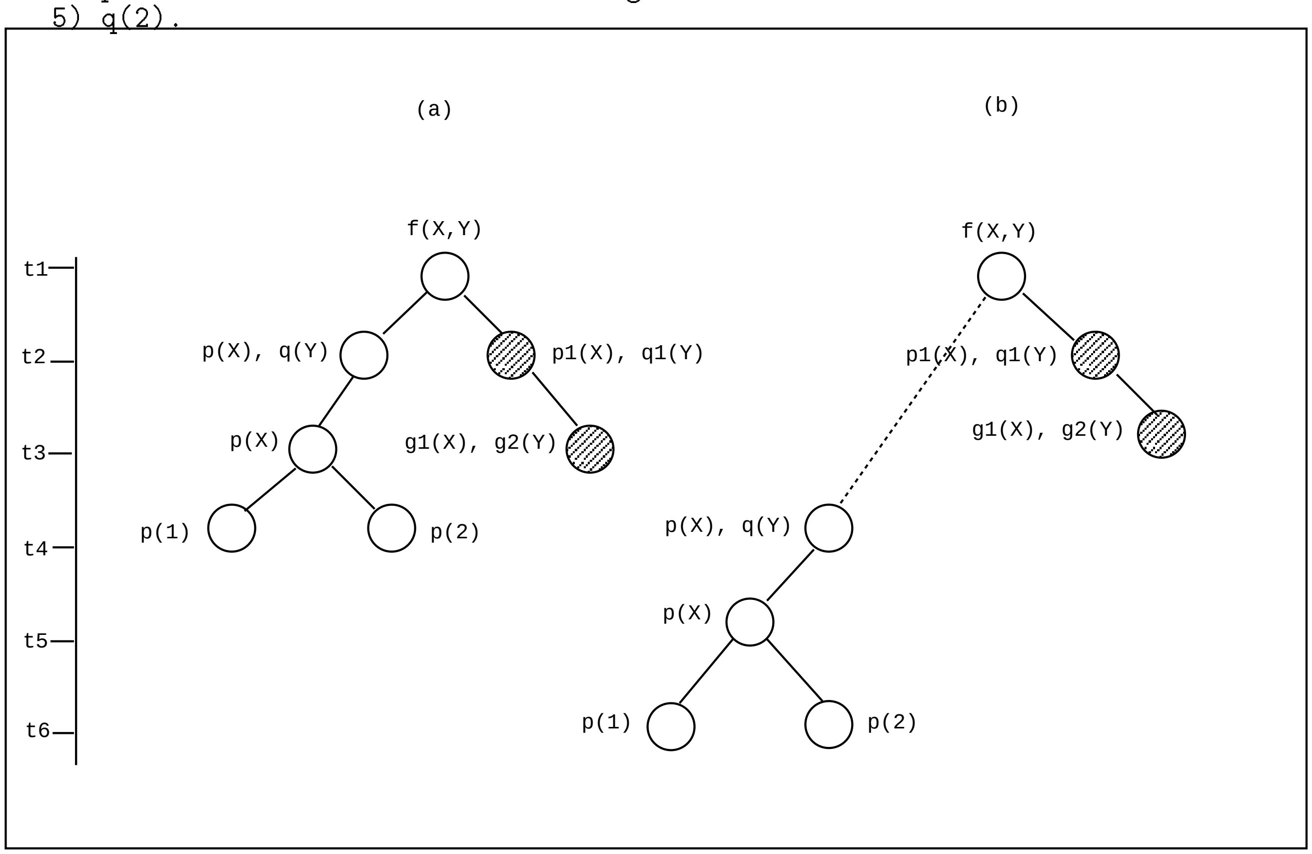

Figure 4: Different trees to execute Example (1) in parallel

Consider the simple trees in Figure 4 based on Example 1. Assume that the execution follows the Andorra computation rule. In Fig 4.(a), the theoretical speedup is $\frac{9}{3}$, i.e. $\frac{number\ of\ nodes}{longest\ path}$. The minimum number of steps to compute all solutions assuming an unbounded number of processors is 3. If we assume a limited number of processors, it does not matter if we execute the two and- goals of one of the dense nodes (shaded) shown in Figure 4 sequentially, because the length of the computation to calculate all the solutions will remain the same and consequently the speedup will remain the same. But if we choose to concentrate all the processors on the right branch (executing and- work), we will increase the length of the computation, transforming the tree of Figure 4.(a) to the tree of Figure 4.(b). The speedup in this case drops to $\frac{9}{5}$.

Our goal is to avoid this kind of situation by allocating the workers fairly to the tasks in the tree.

We can conceive the following four approaches to implement the scheduler strategies. At first, we are going to focus on the first one, as it is simpler and can be immediately applied to Andorra-I. The others are being investigated and should be considered long term research.

1. Traditional Approach

As discussed before, given the theoretical graph of the program (as in Figure 4, for instance), obtained through global analysis, write algorithms that try to keep the speedup as close as possible to the theoretical one (which is governed by the height of the longest path in the theoretical tree), without compromising the cost of task switching. The graph of the program is obtained according to the most general query and contains information about the size of paths, degree of determinacy and priorities of goals. The goal with less priority is the most speculative. We consider to reach or to approximate the size of the longest path because we are concerned about finding all-solutions and in this case the longest path is responsible for the overall speedup of the system. Some experiments can be done in order to build an algorithm to find a first solution. In this case, we should take into account the size of the shortest path.

2. Self organising network

   Given the initial theoretical graph of the program, obtained through global analysis, map processors and tasks into nodes of an artificial neural network (ANN) ([6, 12, 1]).

   For instance, consider example 1 again. For this example we have three branches to be executed, each one with a task size. Each possible combination of task/processor can be mapped as a node of the ANN, so if we have P processors and T branches, we will have P×T nodes in the network. The weight between any two nodes $N_{i,t}$ and $N_{i,q}$ in the same row, i.e. every connection between pairs of nodes that assign the same task to different processors, will be represented by -W (where +W represents the strongest connection) [13]. Whilst the load and the portions of the tree are being modified during the execution, new configurations will be calculated. The weight -W should be calculated from the cost of task switching and priority of execution of tasks.

   Some researchers have been using neural networks to solve dynamic load balancing problems in distributed systems.

3. Simulated Annealing

   Simulated Annealing is an algorithm based on a probabilistic decision of accepting states of computations. It basically consists of two nested loops. The outermost loop governs the number of iterations and checks if the computation is "stable". The innermost loop calculates new states and decides if this state should be output or not based on a probabilistic function. Simulated annealing is also applied to neural networks in order to find global minimum (maximum) of a computation. Ioannidis et al [14] used this algorithm to optimise logical queries, but it is accepted that this can not be the best approach because the algorithm is very slow to converge.

4. Branch and Bound algorithms

   This approach was used by Lipovsky et al [15] to guide the scheduling of or- tasks in a computer architecture dedicated to databases. A heuristic function based on the likelihood of a branch failure is used to guide the search in an or-tree and it is based on a probabilistic model. The scheme is such that the bounds are in two categories: local and global. Local bounds ("weak") are updated during a "session" (sequence of user's queries) and global bounds ("strong") are updated between sessions. Bounds to the start session are taken from the global data. No consideration is made about guiding the search to the and-tree. Moreover all bounds are calculated at runtime. We could generate the starting bound values from global analysis, and adjust them during local and global bounds updates, extending the model to deal with the and-portions of the tree.

5 Conclusion

Our main objective in designing a flexible scheduler for Andorra-I is to allow the system to distribute work among the processors, in such a way that we have several teams when the application demands or-parallelism, and few teams with many workers if the application demands and-parallelism. The results in Table 1 show that we can achieve this goal and get performance close or better than the best produced by Andorra-I with fixed configuration of teams, without having to decide what configuration to use in advance. We believe that much can be gained if we use additional information to guide the top-scheduler strategy. Especially to prevent the "illusion" effect mentioned before, that misleads the top-scheduler when making decisions.

We believe that the problem of task scheduling in and-or trees of logic programming systems can be represented by a mathematical model that makes the task scheduling independent of the model or its implementation. It is sufficient that we identify the parameters involved in task scheduling and relate them in such a way that we can exploit the available and- and or- parallelism optimally.

Acknowledgements

I would very much like to thank David Warren for the discussions and ideas about parallelism and scheduling in logic programming. Also thanks to Rong Yang, Tony Beaumont and Vítor Santos Costa for giving me support for the implementation. I would also like to thank David, Rong, Vítor, Sanjay Raina, Gopal Gupta and Raed Sindaha for reading and correcting earlier drafts of this work.

References

[1] Valmir C. Barbosa and Priscila M. V. Lima. On the Distributed Parallel Simulation of Hopfield's Neural Networks. *Software–Practice and Experience*, 20(10):967–983, October 1990.

[2] Anthony Beaumont, S. Muthu Raman, and Péter Szeredi. Flexible Scheduling of Or-Parallelism in Aurora: The Bristol Scheduler. Technical Report to be published, University of Bristol, Computer Science Department, October 1990.

[3] Per Brand. Wavefront scheduling. Internal Report, Gigalips Project, 1988.

[4] Maurice Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10(2):91–124, February 1991.

[5] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek, and Rick Stevens. Scheduling OR-Parallelism: an Argonne Perspective. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1590–1605. MIT Press, August 1988.

[6] Felipe M. G. França. A Self-Organising Updating Network. In *International Conference on Artificial Neural Networks, ICANN 91, Helsinki*, 1991.

[7] Alan Calderwood. Aurora—Description of Scheduler Interfaces. Internal Report, Gigalips Project, January 1988.

[8] Alan Calderwood and Péter Szeredi. Scheduling Or-parallelism in Aurora – the Manchester Scheduler. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 419–435. MIT Press, June 1989.

[9] Saumya K. Debray, Nai-Wei Lin, and Manuel Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 174–188, June 1990.

[10] I. C. Dutra. Design of a Flexible Scheduler for Andorra-I. Internal Report, ESPRIT Project 2471, PEPMA, January 1991.

[11] Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG*. PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[12] J. J. Hopfield. Neural Networks and Physical Systems with Emergent Collective Computational Abilities. *Proc. Natl. Acad. Sci., USA*, 79:2554–2558, April 1982.

[13] Hung Khei Huang and Valmir Carneiro Barbosa. Static Task Allocation in Heterogeneous Distributed Systems. Technical Report ES-149/88, Federal University of Rio de Janeiro, COPPE/Sistemas, 1988.

[14] Yannis E. Ioannidis and Eugene Wong. Query Optimization by Simulated Annealing. In *SIGMOD - Proceedings of the 1987 International Conference on Management of Data*, pages 9–22. ACM, 1987.

[15] G. J. Lipovsky and M. V. Hermenegildo. B–LOG: A Branch and Bound Methodology for the Parallel Execution of Logic Programs. In *Proceedings of the 1985 International Logic Programming Symposium*, pages 560–567. IEEE, 1985.

[16] 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.

[17] Douglas Pase and Peter Borgwardt. Load Balancing Heuristics and Networks Topologies for Distributed Evaluation of Prolog. Technical report, Imperial College, Department of Computer Science. Available as TR: Parallel Logic Programming 57, Parlog Group.

[18] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I Preprocessor: Supporting full Prolog on the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 443–456. MIT Press, 1991.

[19] Vítor Santos Costa and Rong Yang. Andorra-I User's Guide and reference manual. Technical report, University of Bristol, Computer Science Department, Sept 1990. Internal Report, Gigalips Project.

[20] Raéd Sindaha. Scheduling speculative work in the Aurora Or-parallel Prolog System. Internal Report, Gigalips Project, University of Bristol, March 1990.

[21] Péter Szeredi and Mats Carlsson. The Engine–Scheduler Interface in the Aurora Or–Parallel Prolog System. Technical Report TR-90-09, University of Bristol, Computer Science Department, April 1990.

[22] Péter Szeredi, Mats Carlsson, and Rong Yang. Interfacing Engines and Schedulers in Or-parallel Prolog Systems. In Aarts, E. H. L. and van Leeuwen, J. and Rem, M., editor, *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 439–453. Springer Verlag, June 1991. Lecture Notes in Computer Science 506.

[23] Evan Tick. Compile Time Granularity Analysis for Parallel Logic Programming Systems. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*. ICOT, 1988.

[24] David H. D. Warren. The Andorra model. Presented at Gigalips Project workshop, University of Manchester, March 1988.

[25] David H. D. Warren, Oct 90. Personal Communication.

[26] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*. MIT Press, August 1988.

[27] Rong Yang. Programming in Andorra-I. Internal Report, Gigalips Project, August 1988.

[28] Rong Yang. Solving simple substitution ciphers in Andorra-I. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 113–128. MIT Press, June 1989.

[29] Rong Yang, Vítor Santos Costa, and David H. D. Warren. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 825–839. MIT Press, 1991.