Strategies for Scheduling And- and Or-Work in Parallel Logic Programming Systems

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

Abstract

Recent parallel logic programming systems, such as Andorra-I, can exploit multiple forms of parallelism, and- and or-parallelism, in logic programs. In order to effectively exploit both kinds of parallelism, such systems must address one problem not yet tackled in logic programming systems, namely how to distribute resources between two different kinds of parallel work, and- and or-work. This paper proposes solutions to this problem. We present two *dynamic* scheduling strategies to distribute and- and or-work in parallel logic programming systems. One strategy, *work-based*, guides its decisions by looking at the amount of current and- and or-work available in an application during execution. The other strategy, *efficiency-based*, is designed around the intuition that workers should spend most of their execution time doing useful work, i.e., performing reductions. Results show that both strategies work reasonably well when both kinds of parallelism are available. Both strategies could be applied to other parallel logic programming systems that aim at exploiting both and- and or-parallelism.

1 Introduction

There are two main sources of parallelism in logic programming, namely or- and and-parallelism. Or-parallelism is exploited when several alternatives to a goal are executed in parallel. And-parallelism is exploited when two or more goals are executed 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.

When allowing and-goals to proceed in parallel, the logic programming system needs to distribute parallel and-goals among processors. This job is usually done by an *and-scheduler*. When allowing several alternative clauses to proceed in parallel, the system needs to distribute alternatives among the processors. This task is usually performed by an *or-scheduler*. There are several and-scheduling techniques implemented for scheduling and-parallel

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

work in systems that exploit and-parallelism only, such as &-Prolog [12] and commited-choice languages systems, such as Parlog [4] and FGHC [22]. There are also several or-scheduling techniques implemented for systems that exploit only or-parallelism, such as the Aurora [3, 20] and Muse [1] systems.

In order to achieve the most parallelism from the applications, recent parallel logic programming systems try to exploit multiple forms of parallelism, both and- and or-parallelism. Until now, only the problems of and-scheduling alone and of or-scheduling alone have been tackled by the parallel logic programming researchers, in the context of systems that exploit only a single form of parallelism. When allowing both kinds of parallelism to be exploited, the system needs to deal with an extra problem, namely how to divide the processors effectively between and- and or-work. This is a new and hard problem to be solved in parallel logic programming systems.

The main contribution of this research is to present a solution to this problem in the context of a parallel logic programming system that exploits both and- and or-parallelism, Andorra-I [18, 26]. We propose and compare two *dynamic* scheduling strategies based on two different heuristics to distribute processors between and- and or- tasks. One strategy, *work-based*, allows migration of processors depending on the task sizes estimated during execution time. In an earlier paper [6] we described briefly the work-based strategy and showed some results for a very small number of processors and different applications, for the interpreted-based Andorra-I. In this work we give a more detailed and comprehensive description of the work-based strategy and show results for a larger number of processors using the compiler-based version of Andorra-I.

The other strategy, *efficiency-based*, moves processors from one task to another depending on their efficiency, where efficiency is measured as the percentage of time spent doing useful work over total execution time. Although this strategy is rather simple and could be applied to parallel systems other than logic programming systems, we do not know of any other scheduler that employs a strategy based on performance monitoring. Most of the scheduling strategies reported in the literature are applied to scientific applications that have a very regular and predictable computational task structure. Our strategies are dynamic and applied to irregular and less predictable computations that are typical of logic programs. We implemented both strategies in the Andorra-I parallel logic programming system and evaluate both strategies comparing results for a range of applications, and varying numbers of processors.

The paper is organised as follows. Section 2 describes the Andorra-I system. Section 3 presents the motivation and justification for using scheduling strategies to distribute and- and or-work, and describes the two dynamic strategies proposed. Section 4 describes the benchmark applications, and presents the performance evaluation of both strategies in Andorra-I, for the benchmark set with varying numbers of processors. We also study two of the applications in more detail. Finally, section 5 draws some important conclusions of this work and presents alternative solutions and future work.

2 A Brief Description of Andorra-I

In this section we describe the parallel logic programming system that is the subject of the scheduling strategies: Andorra-I [18, 26]. Andorra-I is based on the Basic Andorra Model [24]. To the best of the author's knowledge, Andorra-I is the first parallel logic programming system that exploits both and- and or-parallelism, and yet run real world applications with significant parallel performance. This is the main motivation to use such a system to show the experiments with scheduling both and- and or-parallelism.

Andorra-I employs a very interesting method of exploiting and-parallelism in logic programs, namely to execute *determinate* goals first and concurrently, where determinate goals are goals that match at most one clause in a program. Thus, Andorra-I exploits determinate dependent and-parallelism. Eager execution of determinate goals can result in a reduced search space, because unnecessary choice-points are eliminated. The Andorra-I system also exploits or-parallelism that arises from the non-determinate goals. Its implementation is influenced by JAM [5] when exploiting and-parallelism, and by Aurora [15] when exploiting or-parallelism. A processing element that performs computation in Andorra-I is called a *worker*. In practice, each worker corresponds to a separate processor. Andorra-I is designed in such a way that workers are classified into *masters* and *slaves*. One master and some or no slaves constitute a *team*. Each master in a team is responsible for creating a new choice-point, while slaves are managed and synchronised by their master. Workers in a team cooperate with each other in order to share available and-work. Different teams of workers cooperate to share or-work.

The Andorra-I system consists of a preprocessor, responsible for generating determinacy code and the sequencing information necessary to maintain the correct execution of Prolog programs, an engine, responsible for the execution of the Andorra-I programs, two different scheduler components, the and-scheduler and the or-scheduler, and the *reconfigurer*. The or-scheduler is responsible for finding or-work, i.e., an unexplored alternative in the or-tree. The and-scheduler is responsible for finding eligible and-work, which corresponds to a goal in the run queue (list of goals not yet executed) of a worker in the same team.

Finally, the reconfigurer, the main subject of this paper, is responsible for arranging the workers into teams in a way that allows both and- and or-parallelism to be freely exploited **when** they are available.

We implemented two reconfiguring strategies in the Andorra-I system and studied how both strategies affected the system's performance. In the next section we describe both strategies and present briefly their implementation in Andorra-I.

3 The Dynamic Scheduling Strategies

The main motivation of having a reconfigurer is to allow a parallel logic programming system to be more flexible when exploiting both kinds of parallel work: and- and or-. Before the reconfigurer was provided in Andorra-I, users had to choose a static arrangement of workers into teams that would provide the best compromise for exploiting the available and- and or-parallelism. Static arrangement of workers into teams might not always produce the best possible speedup achievable. This is clearly seen if we have an application that has and- and or-work in distinct phases. The user can arrange workers in only one team to exploit the and-parallelism, or can arrange workers as masters to exploit or-parallelism. However, no single static configuration given by the user would be able to solve the problem optimally. The reconfigurer would be able to produce a better allocation by dynamically dividing workers into teams so as to produce a much better execution time. We therefore can handle programs that present different sources of parallelism without manual adjustments of reconfigurations for different applications. An *optimal* scheduling algorithm, guaranteed to maintain the best configuration taking all overheads into account, is not feasible. We propose strategies that are based on heuristics. These strategies aim at effectively distributing and- and or-work among the workers, trying, at the same time, to reduce the overall overhead and idle time of the system.

Next, we describe the two strategies to distribute and- and or- work. In order to understand both strategies, it is important to know that the design of the reconfigurer takes into account the fact that there already exist distinct schedulers to deal with and- and or-work. It is out of the scope of this paper to discuss existing and- and or- scheduling techniques. Also, it is important to know that the reconfigurer is called whenever a slave runs out of and-work in its team or a master runs out of or-work.

3.1 The Work-Based Strategy

As the name says, the work-based reconfigurer makes its decisions depending on estimates of the sizes of and- and or-work available in the execution tree. The intuition behind the work-based strategy is that workers will move to places where the work is available to be executed. If the current balance between and-work and or-work available persists during execution, then this strategy is likely to behave well. Otherwise, this strategy may incur too much reconfiguring overheads.

It is important to distinguish two kinds of ideal measures for work in an Andorra-tree. One is the granularity of a determinate phase in terms of estimated number of reductions performed by one processor, in cooperation with other processors. We call it the *parallel depth* of the tree. It corresponds to the number of time steps taken to execute the program with perfect and-parallelism. This kind of measure is important because it gives an estimate of the granularity size of the work to be performed by a worker in a team. The other kind of work is the amount of *parallel* work available in the tree, parallel and-work or parallel or-work. In an Andorra computation, the size of the and-parallelism is given by the width of the determinate phases, and the size of the or-parallelism is given by the number of alternatives still left open. These measures are important because it is helpful to know the maximum number of workers necessary to share the parallel and-work or the parallel or-work.

Granularity information, obtained from compile-time analysis, would help to produce the estimators for sizes of work. Researchers have been working in this direction [27, 14, 8], and the literature reports some improvements in the area. But for many cases it is still difficult to have good estimators. Therefore a reconfiguring strategy should rely on dynamic information, aided, whenever possible, by good estimates of sizes of work. As we do not use any kind of compile-time granularity analysis in Andorra-I to estimate sizes of tasks, our estimates are taken at run-time. Each parallel work unit is considered to have a lower bound value of 1, meaning that one parallel work unit will always produce 1 reduction, i.e., in practice we use a parallel depth of 1. So, one choice-point is considered to produce 1 reduction and one parallel goal is considered to produce 1 reduction. The amount of and-parallel work, i.e., the width of a determinate phase is approximated to the number of goals existing in the run queues of goals, and the amount of parallel or-work is approximated to the number of choice-points existing in the execution tree.

The reconfigurer has two main goals in Andorra-I: (a) to decide when an idle worker will change its type (master/slave), i.e, what kind of work is preferable: and- or or-, and (b) to choose the preferred team for a slave.

In order to find the preferable kind of work *currently* on the tree, the reconfigurer takes, from the and-scheduler, the total number of goals available in the run queues of all teams as an estimate of parallel and-work. The reconfigurer also takes, from the or-scheduler, the total number of choice-points in the execution tree as an estimate of parallel or-work.

In order to choose the preferred team the reconfigurer takes the ratio between sizes of run queues in the team and number of workers in the team. The preferred team is the one that has greater ratio i.e., the one that has greatest *load*, i.e., more amount of work per worker in the team. If we have less workers than parallel work units (goals in the run queue), the parallel depth will increase and, therefore, the and-work that, in theory, could be executed in parallel, is considered to be executed sequentially. The granularity of work for each worker in this case, increases.

There is a cost involved in moving a worker from one kind of work to another. Therefore, we define the *and-threshold* as the estimated number of reductions that makes it worthwhile a worker to move to an and-task. We also define the *or-threshold* as the estimated number of reductions that makes it worthwhile a worker to move to an or-task. This means that if the and-threshold has value N and the current number of reductions to be performed in a team is below N, then the reconfigurer will not take any action.

If the worker requesting work is a *slave*, then, it will *move to another team*, either if there is no or-work available (with and-work available) or if there are both kinds of work and the amount of and-work is considered to be greater than the or-work available. The slave *becomes a master* either if there is no amount of and-work available (with or-work available) or if there are both kinds of work and the amount of or-work is considered to be greater than the amount of and-work.

If the worker requesting work is a *master*, then, it will *become a slave* either if there is no or-work available (with and-work available) or if there are both kinds of work and the amount of and-work is considered to be greater than the amount of or-work.

In order to avoid too eager reconfiguration of workers, we also maintain a counter that controls when a change is to be made. Whenever a worker calls the reconfigurer, it checks the counter. If the value of the counter is below a threshold, it increments it and does not take any further action. If the counter is above the threshold, the reconfigurer will decide if the worker moves to and- or or-work.

If a worker is chosen to become a master, the best or-task for the new master will be found by the or-scheduler according to the or-scheduler strategy used. If a slave is chosen to move to another team or a master becomes a slave, the and-scheduler will find an and-task for the slave in its new team.

3.2 The Efficiency-Based Strategy

The second approach we use to address the dynamic scheduling of and- and or-work does not attempt to estimate amounts of work available in the execution tree. Instead, it relies on the previous efficiency history of the processor.

The intuition behind this strategy is that processors should be busy doing useful work most of the execution time. In order to achieve this, instead of moving workers according to crude estimates of the amount of current work, the strategy relies on previous observation of work performed. Workers check their efficiency and only make a move when they become inefficient. How do we know when a worker is inefficient? We can measure the efficiency of a worker by monitoring the percentage of time the worker is busy over a period of time. The idea is simple: we keep a record of the time a worker is busy by accumulating the time it spends performing reductions. Every so often, the scheduler monitors whether the worker is maintaining its efficiency, measured as a percentage of busy time over the last period of time since the worker became idle, i.e., ran out of work. If the worker's efficiency is below a percentage threshold, the scheduler considers a movement of the worker. A worker can make a certain number of moves: become a slave and move to a team; become a master and start a new team; and move from one team to another.

A worker can also be released to the system if its efficiency drops below a threshold (This strategy is used by the flexible Bristol or-scheduler, in the Aurora system. We have not implemented it yet). The movements for a worker depend on the cost to make the move. In this strategy, the movement chosen by the reconfigurer is always the one that is expected to incur the lowest cost. This heuristic seems reasonable, because if every worker tries to minimise its idle time and tries to make lower cost movements, the overall overhead of reconfiguring will probably be reduced. Table 1 shows the migration costs (in milliseconds) averaged over all the applications and with all numbers of processors measured, taken from a study of Andorra-I's performance. The `reductions` row gives an approximation of the cost of each migration measured as average number of reductions per second (or logical inferences per second, LIPS). The columns `M`→`S`, `S`→`S` and `S`→`M` correspond, respectively, to the movements: change a master to slave, move one slave from one team to another and change a slave into a master. Therefore, it takes twice as much time for a slave worker to become a master and settle a new team than moving between teams. Whenever slaves or masters change their status, their movements are initialised and they can start again a new life cycle.

	M→S	S→S	S→M
time (ms)	2.136	2.280	5.167
reductions	3.2	3.42	7.75

Table 1: Average Migration Costs

In order to keep track of the worker performance, we use one of the functions of the micro-second interface implemented to measure the performance of the Aurora parallel Prolog system with the Manchester scheduler [21], and adapted to work with Andorra-I.

4 Results and Discussion

In this section we investigate how the work-based strategy and efficiency-based strategy behave for a wide range of benchmarks that vary from the ones that have a single form of parallelism to the ones that contain both forms. Due to lack of space, we do not present all benchmarks, but only a representative set of the whole benchmark set used. Other results can be found in [7]. The benchmarks are classified into four groups, depending on wether the parallelism is high or low, and wether it is mainly and-parallelism or or-parallelism. For brevity, we call these groups "high and", "high or", low and", and "low or". The benchmarks are described below. Two of them have been described before in an earlier paper [26]. For the sake of making this paper more complete we describe them again. Some of the benchmarks were written only to validate the reconfigurer. Others are real world applications.

•`detndet`: This program starts with a determinate phase followed by a non determinate phase that contains 180 alternatives. It contains mainly or-parallelism with a little amount of and-parallelism.

•`ndetdet`: This program starts with a non determinate phase followed by a determinate phase. It presents very much or-parallelism and a reasonable amount of and-parallelism.

•`bqu6, bqu8`: This is a queens program written using a Pandora technique [2]. The boards sizes tested were 6x6 and 8x8. The program has mainly or-parallelism with little amount of and-parallelism.

•`cross11`: This is a program that solves crossword problems according to the constraint technique used by van Hentenrick [23]. The crossword program used to run in Andorra-I has a crossboard of 12x12. This program presents little and-parallelism and little or-parallelism, mainly due to the size of the crossboard.

•`road_markings`: This is a program to extract road markings from real world vision pictures that have been processed to identify visible objects. It was developed by Hluchoweckyj [13] from the vision research group, in the department of Computer Science at the University of Bristol. This program has mainly and-parallelism.

•`flypan2`: This is a version of a naval flying programme generator, initially developed by Software Sciences in the University of Leeds for the Royal Navy. It is an example of a real life resource allocation problem. The program has both or-parallelism, arising from the different possible choices, and and-parallelism, arising from the parallel evaluation of different constraints.

•`bcnet`: This is another real world application developed by British Telecom [17] in the course of a collaborative project with Bristol University. The version of the program used as a benchmark uses the Andorra-I constraints facilities [9]. This program has both and- and or-parallelism.

All results were collected on a Sequent Symmetry machine, under DYNIX version 3.2.0 running Andorra-I up to 15 processors. Results are shown in figures 1, 2, 3, and 4. All programs were started with an initial configuration of `w` workers in a single team. Each program was run 10 times. We ignored the best, worst and some anomalous times and averaged the remaining times to produce speedups.

Programs that do not contain much exploitable parallelism, shown in figures 2 and 4, benefit equally from both strategies, showing that Andorra-I with the reconfigurer still can achieve speedups for programs that have little parallelism. Programs high or, shown in figure 1, benefit more from the efficiency-based strategy.

For the programs that contain more and-parallelism, shown in figure 3, there is not much to choose between either strategy.

It is important to note that all results just shown are similar to or bet-

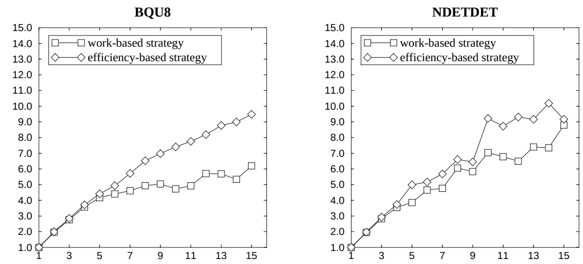

Figure 1: Benchmarks with high or

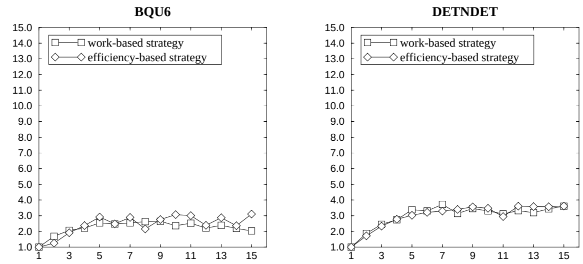

Figure 2: Benchmarks with low or

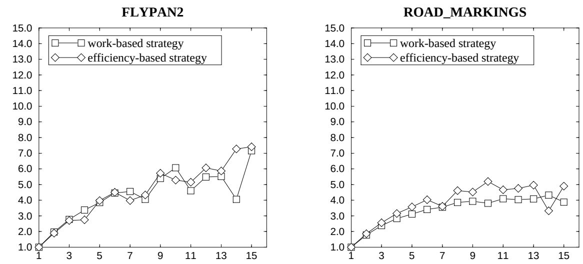

Figure 3: Benchmarks with high and

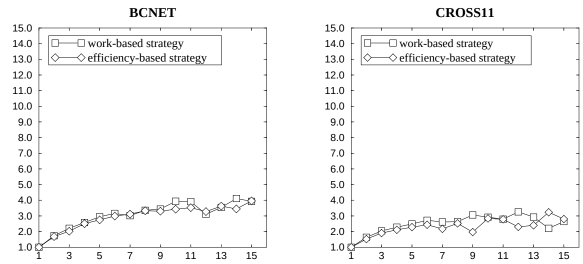

Figure 4: Benchmarks with low and

ter than any fixed configuration of workers adjusted manually. We cannot discuss this issue here due to lack of space. More results comparing the two dynamic strategies with the fixed configuration can be found in [7].

4.1 Case Studies

In order to better illuminate the difference between the two strategies, in this section we study their performance in more detail by showing how the system spends its time in different activities. We show the results for two benchmarks that have a good amount of parallelism, and well known tree structures. They were shown in figure 1. The first one, `bqu8`, presents significant amount of or-parallelism with little amount of and-parallelism while the second one, `ndetdet`, presents both kinds of parallelism, but mainly or-parallelism.

As is clearly shown in figure 1, the speedups for the program `bqu8` using either strategy start to diverge above 6 workers. When we increase the number of processors, the efficiency-based scheduling strategy scales better than the work-based strategy allowing an improvement in average speedups of about 44% over the work-based strategy, from 6 to 15 processors. This happens because the work-based strategy is too eager to reconfigure incurring unnecessary overheads, which do not lead to a better work distribution. This phenomenon happens mostly for programs that have mainly or-parallelism and a little and-parallelism. The work-based strategy knows the estimates for the sizes of the work available, but does not know exactly if the application has more or-parallelism than and-parallelism. As the number of processors increases, the grain size of the tasks decreases and workers start to become idle more often. While the efficiency-based strategy applies almost no reconfigurations in 10 runs with 15 processors, the work-based strategy produces 517 reconfigurations! (for the best run out of 10). So, the efficiency-based strategy also restrains workers from doing unnecessary reconfigurations.

The behaviour of the two strategies for this application can be better studied by looking at how the system is spending its time in different activities. These activities are:

- (T) *reconfiguring*: time spent doing reconfigurations;
- (O) *or-scheduling*: time spent in the or-scheduler (looking for or-work, choosing worker, moving on the tree etc);
- (A) *and-scheduling*: time spent looking for and-work;
- (W) *working*: time spent performing actual reductions (determinacy code or Prolog reductions);
- (I) *idling*: time spent in events like waiting for a signal from a worker, synchronisation between workers etc.

The graphs shown in figure 5 show, for each strategy, the accumulated time spent, as a percentage of the sequential time, for each activity summed over all the workers. For example, the top left graph shows accumulated time for each set of workers (1,2,...,15) for the activity reconfiguring, that is: $\sum_{i=1}^{P} T_{p_i}$, where T_{p_i} is the total time spent by processor p_i in the reconfigurer; and so on (for application `bqu8`). The bottom right graph shows the sum $\sum_{i=1}^{P} W_{p_i}$ that should be approximately the same as the sequential time to execute the same application, where W_{p_i} is the total time that processor p_i spends performing reductions, i.e., doing useful work.

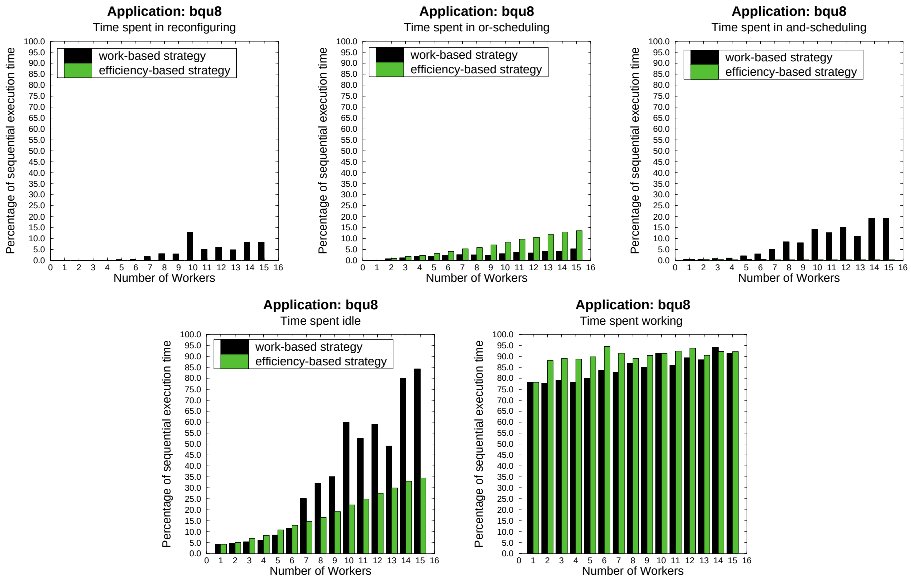

Figure 5: `BQU8`: ACCUMULATED TIME SPENT ON SEVERAL ACTIVITIES

In the graph that shows the time spent in reconfiguring, we can observe that the accumulated time spent in doing reconfigurations is much less when we use the strategy based on efficiency of workers. This reflects on the time spent on the other activities. Workers spend more time in and-scheduling or in or-scheduling if they run out of work. When using the work-based strategy, the time spent in and-scheduling is higher than the time spent in and-scheduling when using the efficiency-based strategy. This happens because there is little and-parallelism in this program, but some of the slaves are running out of and-work for being reconfigured as slaves, with the work-based strategy. On the other hand, the time spent in or-scheduling, when using the efficiency-based strategy, increased with respect to the time spent in or-scheduling when using the work-based strategy. The ratio between the total time spent looking for and-work with the work-based strategy and the total time spent looking for or-work with the efficiency strategy is almost 2. This means that probably idle workers are spending twice as much time running out of work with the work-based strategy than with the efficiency-based strategy. Although the time spent in or-scheduling is higher for the efficiency-based strategy, this strategy produces better speedups than the work-based strategy. In other words, workers become inefficient whenever they are working as slaves, because there is very little amount of and-work to be performed. They are more efficient behaving as masters, even if they spend some time looking for or-work. If we look at the graphs in figure 5, we can observe that with the efficiency-based strategy, workers spend much more time in the or-scheduler than in the and-scheduler.

Another interesting fact is that the amount of idle time is drastically reduced when we use the efficiency-based strategy. The bottom left graph in figure 5 shows how workers have their idling time reduced compared to the work-based strategy.

The bottom right graph shows the accumulated time spent by workers doing actual work, i.e., performing reductions. It is important to note that this program has no speculative work, which means that the accumulated time spent doing work in parallel should be equivalent to the amount of time spent on working sequentially. The graph shows some variation in time spent in performing work. However this variation is introduced partly by measuring errors in the system. Another factor that can also contribute for this variation is synchronisation between workers (e.g., locking and waiting for work).

The other application `ndetdet` is a little more complicated to schedule and we concentrate on explaining it in more detail. The application is a very simple program that contains and- and or-parallelism (the speedup for and-parallelism only is about 2, and the speedup for or-parallelism only is about 8.5, in 10 processors). A choice-point is created with 10 branches. Each of these 10 branches create another choice-point with 4 alternatives. Three of these alternatives (the rightmost ones) fail. Only the leftmost branch of each set of 4 succeeds and gives rise to a reasonable amount of and-parallel work.

If we allow dynamic reconfiguration, the best theoretical speedup for 15 workers can be 15, if masters are fixed on the 10 branches and the slaves work half of their time on each branch with its master. In this case, the master and its slave would produce and-parallel speedups of 1.5 for each branch. In practice, this is very unlikely to happen in a completely dynamic strategy, unless we limit the execution of a worker by the depth of the computation[†]. If we consider the slaves working full time in each branch the speedup is still 10. Figure 1 shows the speedups achieved by this application with up to 15 processors. We can observe that from 10 to 15 workers inclusive, the speedups produced by the efficiency-based strategy fall between 9 and 11, which is quite good, given the intrinsic overheads of the parallel system. From 2 to 5 workers inclusive, the speedups should be theoretically the maximum possible for each number of workers. Again the results are quite good, for both strategies. From 5 to 9 workers the efficiency-based strategy

[†]Thanks to David Warren for pointing me out this possibility.

begins to outperform the work-based strategy. The efficiency-based strategy manages to reach speedup of between 6 and 7 for 9 workers. This is below the best achievable (9) by a dynamic strategy, and coincides with the best speedup achievable with a fixed configuration, but is still good considering the overheads involved in the system.

The variation in time spent in different activities for this program is not so smooth as for the `bqu8` program due to the complexity of allocating workers well for this kind of program. Figure 6 shows how the system spent the time in different activities. The time spent in reconfiguring for this application is less than 0.5% of the execution time. That is the reason why its corresponding graph is presented in a different scale. Because the parallelism in this application can be better exploited for certain numbers of processors, i.e., this is an artificial example with inner regularities, the graphs look like step functions.

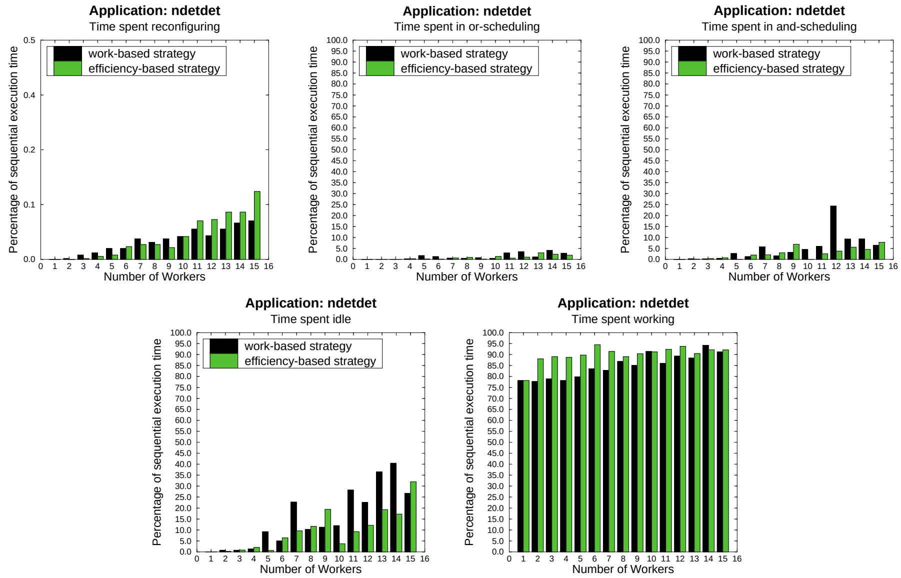

Figure 6: NDETDET: ACCUMULATED TIME SPENT ON SEVERAL ACTIVITIES

5 Conclusions

In this work we presented two dynamic scheduling strategies for Andorra-I that aim at reconfiguring workers in order to exploit and- and or-parallelism efficiently. Adopting a dynamic reconfiguration of workers gives two important advantages. Firstly, a static configuration requires user intervention and knowledge about the nature of parallelism existing in the application. Secondly, a static configuration is bound to be suboptimal in many cases, without achieving the best possible result.

The first strategy described, work-based, relies on assessing the amounts of work existing in the teams or in the choice-points to decide when to move a worker. The efficiency-based strategy makes decisions by monitoring the efficiency of a worker measured as percentage of time spent busy over the execution time. The advantage of the work-based strategy is that if the amount of work taken by the reconfigurer is an exact measure of the actual work to be done, it works fine, but if the estimate of work does not correspond to the actual amount of work to be done, it may not work satisfactorily. Moreover when we add more processors the work grain size reduces and the scheduler ends up doing more reconfigurations than it needs. Also the crude measures of and- and or-work can cause unnecessary reconfigurations, sometimes incurring too many task switches, as was shown by program `bqu8`. This strategy also uses various parameters which means that the parameters values need to be carefully chosen by the user to achieve best performance. The efficiency-based strategy appears to scale up better producing reasonable speedups when we add more processors to the system. By allowing workers to monitor their efficiency, the accumulated idle time and total number of reconfigurations are significantly reduced.

The efficiency-based strategy seems to be a general solution for dynamically scheduling and- and or-parallelism in parallel logic programming systems, given that one has a reasonable knowledge of what are the overheads and performance of the system. It appears that the ideas presented in this paper could be extended for other systems such as EAM [25], ACE [10], PBA [11], ParAKL [16], and Prometheus [19], that aim at combining and- and or-parallelism.

The efficiency-based strategy is at an experimental stage. We believe that speedups for some programs can be improved by tuning the strategy and being less conservative on monitoring efficiency of the workers. It also remains to know if other changes of parameters in the work-based strategy would affect the performance in order that we could choose between one or other strategy for the programs where both strategies produced similar results.

Another important issue tackled with the design of the reconfigurer is the need to control resource utilisation in a parallel multi-user environment by releasing and recruiting workers according to the demand of work. This issue is not presented in this paper, but it is discussed in an associated Ph.D. thesis [7].

Acknowledgements

The idea of monitoring efficiency of workers was born during one of the discussions I had with David Warren. The author is indebted to Vítor Santos Costa, Rong Yang and Tony Beaumont, for their invaluable help and discussions. Special thanks are due to David Warren for encouraging me to carry on this work, and helping me to improve this paper with many suitable comments and suggestions.

Núcleo de Tecnologia Educacional para a Saúde, NUTES/UFRJ allowed me to leave to carry out this research. I am on leave from Programa de Engenharia de Sistemas e Computação, COPPE-Sistemas/UFRJ, Caixa Postal 68511, CEP 21945, Rio de Janeiro, RJ, Brasil.

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] Reem Bahgat. *Non-Deterministic Concurrent Logic Programming in Pandora*, volume 37. World Scientific, Singapore, 1993. Series in Computer Science.

[3] Anthony Beaumont and David H. D. Warren. Scheduling Speculative Work in Or-Parallel Prolog Systems. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 135–149. MIT Press, June 1993.

[4] J. A. Crammond. Scheduling and Variable Assignment in the Parallel Parlog Implementation. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 642–657, October 1990.

[5] J. A. Crammond. The Abstract Machine and Implementation of Parallel Parlog. Technical report, Dept. of Computing, Imperial College, London, June 1990.

[6] Inês de Castro Dutra. A Flexible Scheduler for Andorra-I. In A Beaumont and G. Gupta, editors, *Lecture Notes in Computer Science 569, Parallel Execution of Logic Programs*, pages 70–82. Springer-Verlag, June 1991.

[7] Inês de Castro Dutra. *Distributing AND- and OR-Work in the Andorra-I Parallel Logic Programming System.* PhD thesis, University of Bristol, Department of Computer Science, 1994. Ph.D. thesis, in preparation.

[8] Pedro López García and Manuel Hermenegildo. A Technique for Dynamic Term Size Computation via Program Transformation. Research Report, Facultad de Informática, Universidad Politécnica de Madrid, TR CLIP 8/93.1(94), March 1994.

[9] S. Gregory and R. Yang. Parallel Constraint Solving in Andorra-I. In *Proceedings of the 1992 International Conference on Fifth Generation Computer Systems*, pages 843–850. ICOT, 1992.

[10] Gopal Gupta and M. V. Hermenegildo. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *ICLP'91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, Computer Science Department, University of Bristol, June 1991.

[11] Gopal Gupta and Vítor Santos Costa. And-Or Parallelism in Full Prolog with Paged Binding Arrays. In *LNCS 605, PARLE'92 Parallel Architectures and Languages Europe*, pages 617–632. Springer-Verlag, June 1992.

[12] Manuel V. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-parallel Execution of Logic Programs. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the Fourth International Conference, Volume 2*, pages 556–575. The MIT Press, 1987.

[13] Lydia T. Hluchoweckyj. *Recognising Road Marking Structures for Autonomous Vehicle Navigation using Computer Vision.* PhD thesis, University of Bristol, Bristol, England, 1993.

[14] Nai-Wei Lin and Saumya K. Debray. Cost Analysis of Logic Programs. Technical Report, Department of Computer Science, The University of Arizona, August 1992.

[15] 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.

[16] Remco Moolenaar and Bart Demoen. Optimization Techniques for nondeterministic promotion in the Andorra Kernel Language. In *Proceedings of the Compulog-Net, Madrid*, May 1993.

[17] Dewi Munaf and Brian Tester. And/Or Parallel Programming in Practice. Technical Report, Project 1251:WP 12:1203, British Telecom, November 1993.

[18] Vítor Santos Costa. *Compile Time Analysis for the Parallel Execution of Andorra-I*. PhD thesis, Department of Computer Science, University of Bristol, August 1993.

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

[20] Raéd Yousef Sindaha. *Branch-Level Scheduling in Aurora: The Dharma Scheduler*. PhD thesis, University of Bristol, Department of Computer Science, In preparation, 1993.

[21] Péter Szeredi. Performance Analysis of the Aurora Or-parallel Prolog System. In *Proceedings of the 1989 North American Conference on Logic Programming*, pages 713–732. MIT Press, October 1989.

[22] Evan Tick. Compile Time Granularity Analysis for Parallel Logic Programming Systems. *New Generation Computing*, 7(2,3):325–337, 1990.

[23] Pascal van Hentenryck. *Constraint Satisfation in Logic programming*. MIT Press, 1989.

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

[25] David H. D. Warren. The Extended Andorra Model with Implicit Control. Presented at ICLP'90 Workshop on Parallel Logic Programming, Eilat, Israel, June 1990.

[26] Rong Yang, Tony Beaumont, Inês Dutra, Vítor Santos Costa, and David H. D. Warren. Performance of the Compiler-Based Andorra-I System. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 150–166. MIT Press, June 1993.

[27] X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A. V. S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. Technical Report CIS-TR-91-19, University of Oregon, Department of Computer and Information Science, 1991.