Distributing And-Work and Or-Work in Parallel Logic Programming Systems

Inês de Castro Dutra
COPPE/Sistemas, Federal University of Rio de Janeiro, Brazil
e-mail: ines@cos.ufrj.br

Abstract

In parallel logic programming systems that exploit both and-parallelism and or-parallelism, a problem arises that is how to distribute processors between the dynamically varying amounts of and-work and or-work that are available. Solutions have been reported for distributing only or-work, or distributing only and-work, but the issue of distributing processors between both kinds of work has not yet been addressed. In this work we discuss the problem of distributing and-work and or-work in the context of Andorra-I, a parallel logic programming system that exploits determinate and-parallelism and or-parallelism. We describe dynamic scheduling strategies that aim at efficiently distributing processors between and-work and or-work, and compare their performance with the performance produced by a static scheduling strategy, for a wide range of benchmarks.

Keywords: and-or scheduling, and-or parallelism, Andorra-I, performance evaluation, logic programming.

1 Introduction

An important problem that remains to be solved in the context of parallel systems, in general, is task scheduling. This problem has been approached in the context of logic programming systems that exploit only one form of control-parallelism, and-parallelism or or-parallelism, but has not been addressed for systems that exploit both forms of control-parallelism. In the most recent implementations, such as Andorra-I [19], PBA [8], ACE [6], ParAKL [11] and Penny [10], and models of parallel logic programming systems, such as the Extended Andorra Model [16] and IDIOM [7], both and- and or-parallelism are exploited in a single framework. The aim of these systems is to achieve the most parallelism from the applications. When allowing both kinds of parallelism to be exploited, the system needs to deal with an extra problem, namely, how to distribute the processors effectively between and-work and or-work. This is a new and hard problem to be solved in parallel logic programming systems.

The main subject of this work is the task scheduling of multiple forms of control-parallelism in the context of the Andorra-I parallel logic programming system. Andorra-I parallel execution produces two kinds of work: and-work and or-work. These two kinds of work have different characteristics in Andorra-I. For example, and-parallel work in Andorra-I can be finer grained than or-parallel work, and its exploitation demands more communication among processors than the exploitation of or-parallel work, because of variable sharing. Therefore we have a problem to solve, that is, how to distribute resources among the and-work and or-work existing simultaneously during the execution. In particular, since the system aims to exploit parallelism implicitly, we were challenged to add to the system a key component that would distribute processors to the and-work and or-work available *dynamically*. We chose the dynamic approach because:

- the degree of and-parallelism and or-parallelism varies with execution time, thus requiring dynamic scheduling decisions;
- logic programs have very irregular computation patterns that are not suitable for static partitioning;
- logic programs are difficult to analyse at compile-time, because the programs generate different computation trees for different sets of queries and size of the input data.

Andorra-I exploits a particular form of and-parallelism that is to execute first and eagerly goals that match at most one clause in the program.

The restriction of executing goals that match at most one clause in the program, i.e., to execute *determinate* goals, makes things simpler and allows for a more intelligent method of search. This method of exploiting parallelism was first envisaged in the Basic Andorra execution model [15].

Andorra-I exploits or-parallelism much in the same way as in the Aurora [9] or-parallel system. The bindings for the conditional variables (variables that need to be unbound on backtracking and were created before the last choicepoint) are stored in binding arrays according to the SRI model [14]. It exploits and-parallelism similarly to JAM Parlog [2].

In Andorra-I, workers are arranged into *teams* that cooperate to exploit or-parallelism. Workers within a team cooperate to exploit and-parallelism. A team is composed of a *master* and some or no *slaves*.

Andorra-I exhaustively executes all available determinate goals in the pool of goals and if there are no more determinate goals, the system chooses the leftmost non-determinate goal to create a choicepoint.

The Andorra-I main components are: a preprocessor, responsible for generating determinacy code and sequencing information necessary to maintain the correct execution of Prolog programs [12]; an engine [19], responsible for the execution of the Andorra-I programs [17, 13], two schedulers, the and-scheduler and the or-scheduler [1], and a *reconfigurer*.

The main objective of the Andorra-I schedulers is the selection of which piece of work to execute next, since we may not have enough processors to execute all work at once. The policy used by the schedulers in Andorra-I is demand-driven, which means that whenever a worker runs out of work, its corresponding scheduler tries to find another piece of available work. It is also preemptive. A worker can give up its current work in favour of another piece of work. A new problem that arises to the system because it exploits both and-parallelism and or-parallelism is: what kind of work to select next, provided that both and-work and or-work are available?

2 Statement of the Problem

Before our work, Andorra-I ran with fixed configuration of workers into teams, set by the user. In that case, the distribution of workers between the two kinds of work was left to the user's hands. This characteristic of the system, besides being inconvenient to the user, also went against the aim of exploiting parallelism implicitly. Moreover, fixed configurations of workers into teams that never change during execution very often produced results far from optimal.

This work concentrates on the problem of what kind of work to choose, and more generally, where to redeploy workers to. The Andorra-I component responsible for doing that is the *reconfigurer*.

The general problem of reconfiguring can be summarised through a simple example. The picture shown in Fig. 1 shows an Andorra-tree, that is composed basically of an Or-tree, whose branches contain some "bushes" that correspond to the determinate phase of execution wherein all slaves inside the team working on that branch are cooperating to execute determinate goals eagerly. When no more determinate goals are available, a non-determinate phase is started with the leftmost non-determinate goal being picked up to generate a choicepoint, and the Andorra-tree being split into branches (thus producing another or-subtree).

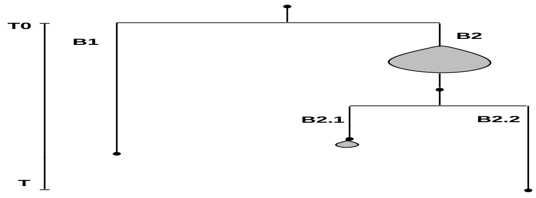

Figure 1: An Andorra-Tree

In Figure 1, at the root node, there is a choicepoint with two alternatives. The first branch (`B1`) has no amount of and-parallelism while the second branch (`B2`) has a reasonable amount of and-parallel work followed by or-parallel work. This or-parallel work is represented by the choicepoint that leads to the alternative branches `B2.1` and `B2.2`. Branch `B2.1` by its turn has a small amount of and-parallel work, while branch `B2.2` has no and-parallel work. Ideally we would like the system to allocate the right number of processors to the branches, without the help of the user. Additionally, the system should finish the job within or close to time `T - T0`.

In this work we are very much concerned with the criteria used by the reconfigurer to *automatically* rearrange workers among the teams and consequently distribute and-work and or-work in a way that parallelism is exploited with a minimum of user intervention and with minimum overhead. If possible we would like also to obtain reasonable speedups close to the best achievable.

As, before our work, the user had to choose a fixed configuration of masters and slaves to exploit the parallelism available, this brought three main drawbacks to the system:

- It was very inconvenient and difficult to use, and it was inconsistent with the aim of exploiting parallelism implicitly.
- For most cases, the user would choose a configuration of workers that would produce results far from being optimal.
- Because of the varying nature of parallelism in some programs, the system would still produce performance below the best achievable, even if the user could (somehow) choose the best fixed configuration.

A dynamic strategy not only solves the problem of making Andorra-I a practical system, but can also allow the system to exploit more parallelism from programs where and- and or-parallelism vary with time. A very simple example is a program whose search tree produces and-parallelism in the beginning of the execution, say to set up a set of constraints, and later produces or-parallelism to search for a solution. No single fixed configuration of reasonable number of workers into teams solves this problem optimally, because the system should be able to configure workers in order that they exploit and-parallelism in the beginning, and later, work independently in each or-parallel branch.

We use two approaches to distribute and-work and or-work in Andorra-I. One, the *work-guided strategy* (WGS), heuristically tries to observe the future of the computation by predicting sizes of work. The other approach, the *efficiency-guided strategy* (EGS), observes the past of the computation trying to guide decisions according to the percentage of time the worker is actually performing work.

3 The Work-Guided Strategy

The rationale behind the work-guided strategy is to make decisions based on estimates of sizes of work. As we do not use any kind of compile-time granularity analysis in Andorra-I to estimate sizes of work, our estimates are taken at runtime and correspond to the current amounts of and-work and or-work available during computation. The intuition behind this strategy is that workers will be redeployed to rich sources of parallel work. If the current balance between and-work and or-work available persists during execution, then this strategy is likely to behave well, because the amounts of work will not vary often, therefore, keeping workers busy without having to reconfigure. If the amounts of work and size of parallelism vary often, this strategy may incur too much reconfiguring overheads.

As the work-guided strategy does not rely on any kind of compile-time analysis, it uses parameters plus estimators for sizes of work. If we were capable of obtaining information about an Andorra computation before executing a program, we could identify at least four measures of work in the Andorra-tree.

The first measure for size of work is the granularity of a branch when it is executed sequentially, that corresponds to the total number of reductions performed by only one processor in a branch. We call it the *sequential depth* (r) of the branch.

The second measure for work that exists in an Andorra-tree is the *width* (w) of a branch, i.e., the number of available and-work units that can be executed in parallel.

The third measure is the granularity of the bush when we exploit perfect and-parallelism, i.e., the distance from a choicepoint to the next choicepoint. We will call it the *parallel depth* (D) of the bush. Goals in a bush or determinate phase have different parallel depths, defined as D_g, where g is a parallel goal. The parallel depth of a bush or determinate phase is associated with the parallel depth of the determinate goal that leads to the next choicepoint. Ideally, we would like to allocate as many processors as there are goals for each level of a bush (determinate phase) of the Andorra-tree, in order that each goal can be executed by one processor.

The last measure for work that we can see in an Andorra-tree is the amount of or-work available in the tree (o). The best estimate for this measure is the number of alternatives left open in the tree, that are non-speculative. We put this restriction of not counting speculative work as part of the or-work, because this would lead to more computation than necessary. However, one could well choose to count speculative work as part of the or-work if this would lead to a faster execution.

The reason why we discuss measures of work in an Andorra-tree is that we designed our work-guided strategy around measures of *sizes* of work in the tree. However the measures we mentioned so far are based on an ideal analysis of an execution tree. If it were possible to obtain all that information about sizes of work before starting the execution of the program, then the distribution of processors could be done statically. Unfortunately, the real world is very different from what we would expect ideally. Therefore, as we do not use any kind of compile-time analysis to give us the precise and correct sizes of work in a computation tree (which is very difficult in itself), we estimate the sizes through parameters and information collected at runtime.

The width (w) of a determinate phase in Andorra-I is given by the size of queues of goals. In that case, the width varies dynamically at each point of the computation. Moreover, the queues of goals may not contain only goals that can be executed in parallel.

The reconfigurer takes, from the or-scheduler, the total number of live nodes (nodes that still have alternatives to be taken) or alternatives in the execution tree as an estimate of or-work. The number of live nodes and the number of alternatives left in the execution tree are two possible estimates for or-work (o) in Andorra-I.

There is a cost involved in redeploying a worker from one kind of work to another. Therefore, we define the *and-threshold* as the number of reductions that makes it worthwhile for a worker to be redeployed to and-work. We also define the *or-threshold* as the number of reductions that makes it worthwhile for a worker to be redeployed to or-work. This means that if the and-threshold has value N and the current number of reductions to be performed in a team (D_t, i.e., the parallel depth of team t) is below N, then the reconfigurer will not take any action. The values for the and- and or-thresholds are taken from a performance analysis of Andorra-I [18].

In order to compare amount of and-work with amount of or-work, which are two different kinds of measure (like meters and calories!), we use a correction factor, so that whenever we compare and-work with or-work we do not compare absolute values, but relative values. In order to compare and-work with or-work we use the correction factor G_a. In order to compare or-work with and-work, we use the correction factor G_o.

Workers scan the run queues to find available and-work. The load of a team is the parallel depth of that team expressed as $\frac{w_t}{R_t} * D_t$, where w_t is the size of the run queues in team t, R_t is the current number of workers in the team, and D_t is the estimated parallel depth for each goal in team `t`.

4 The Efficiency-Guided Strategy

The efficiency-guided strategy uses a heuristic that is different in philosophy from the work-guided strategy. Instead of looking at present and possible predictions of future sizes of computation, it observes the past computation in order to redeploy workers that remain idle in a team.

As a matter of fact, many dynamic and reactive systems use this heuristic of looking at the past computation, as it is usually more reliable than to make predictions of the future or use instant runtime information. The other reason why dynamic systems use past computation is that it is easier to evaluate the behaviour through stochastic methods. This is the case, for example, of distributed network systems.

The intuition behind the strategy we describe in this section is that *processors must be kept busy most of the execution time*. This is a reasonable assumption, since a processor in a parallel system may be performing activities other than performing useful work, i.e., performing reductions. A processor can be spending time idling, synchronising, waiting for another worker, or waiting for work.

There are at least two ways to observe whether a processor is busy most of the execution time:

- The first one is to monitor the number of reductions a worker performs per time unit during the parallel execution and compare with the number of reductions the processor *should be* performing within that time unit. Obviously, we need to know how many reductions a worker should be performing for the given application. This can be taken from the sequential execution of the program.
- The second way is to observe the total time a processor spends busy, performing reductions, over the total execution time.

The first way of looking at the past computation of a processor is very application dependent, since the *lips* rate, i.e., the number of logical inferences per second of an application varies from application to application. This variation is due to several reasons, one of them being the complexity of the abstract instructions executed by the application according to the size of complex structures. Unification, for example, can be a very heavy operation for some applications, and be very light for others.

The second way of observing performance is application independent, and therefore more general. In this second approach we keep track of the worker's performance by monitoring its time performing reductions along the execution. While in a parallel execution, a processor can spend its time by performing reductions, looking for work, or idling. We take the percentage of time performing reductions over the total execution time as a measure of processor utilisation.

Workers in the efficiency-guided strategy are redeployed whenever they are below the expected performance level (this performance level can be the system's default or can be given by the user). If a worker is about to be redeployed, it will try the redeployment with the least cost, but having the chance of navigating through all cost levels. The possible redeployment levels for a worker are: (1) master is redeployed from or-work to or-work (no change); (2) slave is redeployed from and-work to and-work (change teams); (3) master is redeployed from or-work to and-work (becomes a slave); (4) slave is redeployed from and-work to or-work (creates a new team); (5) master leaves the current application and is released to the system; (6) slave leaves the current application and is released to the system; (7) master returns to the current application through a request to the system; (8) slave returns to the current application through a request to the system.

The redeployments 5, 6, 7 and 8 address the problem of efficiency in a parallel multi-tasking environment, by releasing resources when they are being used inefficiently after some period of time, or retrieving resources back from the system when the application needs them. The optimal solution is a trade-off among several factors: the cost to use the resource (processor), the speed the user wants its application to run, the number of resources available in the system, the number of users available in the system, the response-time of the operating system. We have not implemented this feature in the system, but it is definitely an important topic of research where several groups have been working [20, 3].

In Andorra-I, slaves are subject to *slave redeployments* and masters are subject to *master redeployments*. Slave redeployments are (in order of cost): (a) stay as slave; (b) change to another team; (c) become a master; (d) be "dismissed" from its job (i.e., be released to the operating system to be used by another user). The valid master redeployments are: (a) stay as master; (b) become a slave; (c) be "dismissed" from its job. Whenever a slave or a master changes its status, their redeployment field is initialised and they can start a new life cycle. The costs of redeployment are taken from the performance analysis of the Andorra-I system. However they can also be given by the user.

One important implementation aspect of the efficiency-guided strategy is the time interval to check performance. If we choose the time interval to be very short, we may incur a very high overhead. If we choose the time interval to be very long, workers may stay inefficient for a long time.

Another aspect of the efficiency-guided strategy is to choose the length of the time interval to check efficiency i.e., the size of the observed history of the worker. An approach is to be very conservative and remember efficiency since the query started. Another approach is to be less conservative and "forget" some of the past computation. If we use the time since the query started, and the tendency of the system is to stabilise, then we will converge to the right redeployments always. By using the time since the query started in order to monitor efficiency, we can get a more general picture of the history of the computation and redeploy workers with more reliable and stable information. If we use "windows" to check the performance of the worker, because logic programming computations are intrinsically irregular, the right sizes of the windows will depend on a given computation. On the other hand, a conservative choice such as to observe efficiency since the query started may make workers realise they are inefficient very late in the computation. However this choice also prevents workers of doing too many reconfigurations.

The efficiency threshold is application independent and needs to be well chosen in order to allow workers to do the appropriate number of redeployments. If we choose the efficiency threshold to be 100 (in theory, a processor that is not being utilised 100% should be redeployed), the number of task switches increases, because, in practice it is not possible to keep a worker busy 100% of its execution time. If we choose the efficiency threshold to be too low, the workers will take too long to be redeployed, and therefore will stay inefficient most of the execution time. The choice for this parameter needs to be a trade-off between redeploying too eagerly or redeploying less eagerly.

Whenever a worker detects that it is inefficient, it is redeployed according to the cost of redeployment. The reconfigurer always chooses the redeployment of lower cost. If the worker is changing teams or becoming a slave, the reconfigurer looks for the best team for this worker. The best way of doing so is by checking which team is the most efficient, i.e., which team has more efficient workers, where we get the efficiency of each worker in the team and take the average for the team.

If a master or a slave is redeployed to or-work, it is up to the or-scheduler to decide what node to move to.

5 Results

In this section we evaluate Andorra-I with the reconfigurer, i.e., with *dynamic* distribution of and-work and or-work. First, we analyse how Andorra-I with a reconfigurer compares with Andorra-I in the past. Secondly, we show some results for the work-guided strategy and for the efficiency-guided strategy.

Results in this section were collected on a Sequent Symmetry machine with 16 processors, running DYNIX.

More detailed results and description of the benchmarks can be found in [5].

All programs used as the benchmark set were selected according to their degree of parallelism. One group of programs has predominantly and-parallelism, another has predominantly or-parallelism, another has both kinds of parallelism in different phases of the computation, and another has both kinds of parallelism appearing at the same computational phase.

Some of the benchmarks are specially written to test the reconfigurer. Others are real applications used by companies or by academic people. The idea behind writing special programs is to predict the behaviour of the reconfigurer and evaluate closely the scheduling strategies.

In the next tables, benchmarks are classified in: (1) artificial programs, specially written to test the reconfigurer; (2) programs that contain predominantly or-parallelism in decreasing order of amount of or-parallelism, and (3) programs that contain predominantly and-parallelism in decreasing order of amount of and-parallelism.

5.1 Dynamic Reconfiguring

In order to demonstrate the benefits of using a reconfigurer in Andorra-I, we compare the reconfigurer results with several possible fixed configurations of workers representing plausible user choices in the old Andorra-I system. We show that Andorra-I with the reconfigurer, without any user intervention, performs better than the old Andorra-I in likely practical use.

In order to see if the reconfigurer, in particular the work-guided strategy, is performing as well as it could, we evaluate its performance by comparing it with a *target performance*, which is defined as the best performance achievable with any fixed configuration (allowing this to be freely chosen to fit the benchmark and number of processors). We show that the work-guided strategy performs better than the best fixed configuration for most of the benchmarks.

We start by comparing the performance of our reconfigurer with the performance achieved by different versions of the old Andorra-I corresponding to different fixed configurations chosen. This method of comparing Andorra-I with the reconfigurer with the old version of Andorra-I seems reasonable, since we take real results produced in practice with plausible fixed configurations.

There is a very big number of fixed configurations possible. We will limit our comparison to three plausible fixed configurations. As explained before, in Andorra-I, users can specify the number of teams and number of slaves to run a computation, where the slaves are evenly allocated to each team. In order to allow the user to enter a more convenient fixed configuration for all numbers of processors we assume that the system provides a formal way of choosing the configurations. In that case for a number of processors n we will use the following formula, where we only have plausible user choices, assuming a particular weight between and-parallelism and or-parallelism, and with teams having approximately the same size.

$$n^p \textit{ teams with } n^{1-p} \textit{ workers each}$$

The number p corresponds to the proportion of or-parallelism, i.e. the relative weight we want to give to or-parallelism, and it is in the range 0 to 1. We take the least integer approximation for n^p and n^{1-p}, with n being the number of processors. If there are any remaining workers from the approximation to n^{1-p}, they are allocated evenly as slaves to the existing teams, until there are no more remaining workers. This formula allows the user to specify weights for and-parallelism and or-parallelism, and to use a single formula for any numbers of processors.

We consider the following three plausible configurations:

1) give weight 0 to and-parallelism. In this case $1-p = 0$, and $p = 1$. We have n teams of 1 worker each.

2) give weight 0 to or-parallelism. In this case $p = 0$. We have 1 team of n workers.

3) give equal weights to both and- and or-parallelism. In this case $p = \frac{1}{2}$ and $1 - p = \frac{1}{2}$. In that case we have $\sqrt[2]{n}$ teams with $\sqrt[2]{n}$ workers each.

As an example, for 10 workers, the system would set a configuration of a team with 10 masters for weighting 1, 1 team of 1 master and 9 slaves for weighting 2, 3 teams of three masters with the first master having three slaves and the two remaining masters having two slaves each for weighting 3. The size of the teams for different choices gives the desired balance to exploit and-parallelism and or-parallelism.

Table 1 shows speedups achieved at 10 processors with the plausible fixed configurations, and the speedups achieved with the reconfigurer. The three middle columns of the table (with $p = 0$, $p = \frac{1}{2}$ and $p = 1$) give the speedups of the benchmark set for the three plausible fixed configurations, in increasing order of p. The column `Speedup with Reconfigurer` gives the speedups achieved by the reconfigurer. Boldface numbers show the best speedups achieved at different fixed configurations. The benchmarks are presented in the following order: artificial programs, programs that contain predominantly or-parallelism in decreasing order of amount of or-parallelism, and programs that contain predominantly and-parallelism, in decreasing order of amount of and-parallelism. The last row of the table shows the harmonic mean over all speedups. The reason for using the harmonic mean in this context is to find an overall mean performance for all benchmarks, giving equal weight to each individual benchmark.

For our benchmark set, despite the fact that we have reconfiguring overheads with the work-guided strategy, the work-guided strategy is consistently close to or better than the best of the three fixed configurations. The reconfigurer reaches an overall result that is around 55% better (≈ 1.5 times faster) than the best result produced with a single fixed configuration (which is given by the choice equal weights to both and- and or-parallelism). It is interesting to note that $p = \frac{1}{2}$ gives the best performance overall for the old Andorra-I, but is not the best individually for most of the benchmarks.

From the figures shown in table 1, we can summarise the following:

- For computations that contain or-parallelism only, `chat` and `mutest`, the reconfigurer performs similarly to a fixed configuration of n teams with one worker each (choice of user 1), which means that the overhead of reconfiguring slaves into masters is negligible.

Benchmarks	Speedup with weighting, p = 0	$\frac{1}{2}$	1	Target Performance		Speedup with Reconfigurer
ndetdet	2.037	5.913	**8.487**	8.487	*(10M)*	9.831
mixed	0.980	2.642	**3.584**	3.997	*(6M4S)*	4.004
detndet	1.044	**2.052**	2.032	3.012	*(8M2S)*	3.157
bqu8	1.527	3.937	**7.516**	7.516	*(10M)*	4.474
chat	0.666	2.502	**7.334**	7.334	*(10M)*	7.034
cypher	1.147	2.875	**5.922**	5.922	*(10M)*	6.116
mutest	0.584	2.071	**5.959**	5.959	*(10M)*	6.025
bqu6	1.499	2.940	**3.137**	3.950	*(9M1S)*	2.920
flypan2	**6.203**	3.507	1.007	6.203	*(9S)*	6.307
bt_cluster	**6.002**	3.947	1.011	6.002	*(9S)*	5.793
flypan4	**5.079**	3.305	1.062	5.079	*(9S)*	4.961
scanner	**5.009**	4.992	3.226	5.485	*(2M8S)*	5.414
road_markings	**4.887**	2.479	0.965	4.887	*(9S)*	4.269
bcnet	3.226	**3.336**	2.033	3.746	*(2M8S)*	3.942
cross11	**2.666**	1.615	1.112	2.666	*(9S)*	2.917
flypan5	**2.629**	2.506	1.335	2.862	*(2M8S)*	3.382
cross6	**2.529**	2.095	0.964	2.529	*(9S)*	2.260
H.mean	1.656	**2.787**	1.836	4.407		4.258

Table 1: Andorra-I in the past x Andorra-I with the reconfigurer, at 10 workers

- For computations that contain and-parallelism only, `bt_cluster`, the reconfigurer performs similarly to a fixed configuration of workers in a single team (choice of user 2), which means that there was no overhead for reconfiguring during the computation.

- For computations with high degree of parallelism and distinct phases of computation, and-parallel and or-parallel phase, despite the reconfiguring overheads, the work-guided strategy performs similarly or better than the fixed configurations. This is shown for computations `ndetdet`, `detndet`, `flypan2`, and `scanner`.

- For computations that contain mainly one form of parallelism, but with small amounts of the other form, sometimes the work-guided strategy does not perform so well as one of the fixed configuration. This is shown for computations `bqu8` and `road_markings`. The difference is very significant for `bqu8`. In other cases, as for `mixed`, `cypher`, and `flypan4`, the performance obtained with the reconfigurer is comparable or better than the performance obtained with Andorra-I without the reconfigurer.

- For computations with a low degree of parallelism, e.g. `bcnet` and `cross11`, the reconfigurer performs slightly better than any of the fixed configurations. For others like `bqu6`, one of the fixed configurations performs slightly better.

In summary, we can conclude that Andorra-I with dynamic reconfiguration is overall far better than any single fixed configuration, and even on individual benchmarks is generally better than any fixed configuration that the user might plausibly choose. Moreover, this was achieved *automatically* without any user intervention.

After showing that Andorra-I with the reconfigurer performs much better than Andorra-I without the reconfigurer, we intend to evaluate how good is the performance of the work-guided strategy compared with the best performance we might hope to achieve. This study is very important, since we are not only interested in showing the benefits of using dynamic reconfiguration in Andorra-I, but also we are interested in obtaining the best possible performance.

There are several ways that can be used to evaluate the performance of the work-guided strategy. One of them is to use an analytical model to evaluate if the performance of the system corresponds to the optimal model of each computation. Another method is to find the optimal performance of a computation, for limited and unlimited number of processors by simulating the parallel model. Yet another method is to evaluate the performance through pure measurement, i.e., we have two different systems, Andorra-I without the reconfigurer and Andorra-I with the reconfigurer, we collect results from these two different systems and compare both performances.

Our approach consists of evaluating the results by pure measurement by comparing our new Andorra-I system with the old Andorra-I system. Although this method does not evaluate the system with respect to the best optimal achievable speedups, it is still useful as an evaluation method. As we do not have a tool to estimate the performance of the Andorra model or of the Andorra-I system, we limited our study of performance to measurements of performance for different fixed configurations of workers, or in other words, different versions of the Andorra-I system. Our ambitious goal is to achieve at least the best performance achievable with any fixed configuration, which is a difficult target given that the reconfigurer is dynamic (incurs overheads), and sometimes does not make the right choices due to the instant measures taken for the amounts of and-work and or-work. We will call the best performance achievable the *target performance*. We define the target performance as the one produced when we choose the best fixed configuration of workers into teams for each computation at each number of processors. As most logic programs of our benchmark set can run reasonably well with the best configuration of workers into teams, if we can produce results similar or better than the ones achieved by this best fixed configuration, then we are well enough able to say that our objectives are completely satisfied.

Therefore, we compare the performance of the work-guided strategy with what we defined as our target performance. This target performance was determined by running all programs with all possible fixed teams configurations, and getting the best performance for each configuration. As an example, we show in column 5 of table 1 speedups achieved for the best fixed teams configuration of 10 workers for each of our benchmarks. The best fixed configuration is shown between parentheses. For example, program `detndet` achieves its best speedup, at 10 processors, with a fixed configuration of eight masters and two slaves, i.e., eight teams, with two of them with two workers and the remaining six with only one worker.

The overall result shown by row `H.mean` confirms that the work-guided strategy performs similarly to the target performance. For each benchmark individually, the work-guided strategy performs better than the target performance for 10 benchmarks. For the remaining 7 benchmarks, the work-guided strategy does not reach our target performance, although the results are still good, given that it is not an easy task to allocate dynamically the right numbers of workers to the varying parallel work available.

5.2 Evaluation of the EGS

In this section we study the performance of the efficiency-guided strategy when compared with the work-guided strategy and target performances. In the second part, we illustrate the behaviour of the two strategies for two benchmarks chosen from the benchmark set. More detailed studies of the two strategies can be found elsewhere [4, 5].

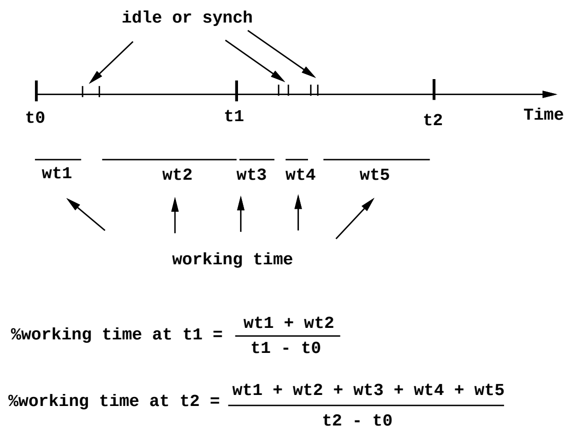

Figure 2: EGS ALONG THE EXECUTION TIME

We discussed before about the efficiency threshold. This is the parameter that controls at what efficiency level a worker must work without having to be redeployed to elsewhere. We discussed that the efficiency threshold needed to be well chosen in order to avoid too many reconfigurations or too few reconfigurations. From our experiments, we found out that the value 50 was the ideal for all benchmarks. This was found to be the ideal compromise between having too many reconfigurations or too few reconfigurations. The value 50 means that a worker will be redeployed if its performance drops below 50%. If its performance, total time working over total execution time, is greater or equal than 50, then the worker remains in its original team. Notice that this number may need to change if we are running the programs in another parallel system. As this number is a percentage of processor utilisation, we believe it is not dependent on the machine architecture we are using.

Another issue discussed in the previous chapter was the time interval used to measure efficiency. Our implementation takes efficient time since the query started. Figure 2 shows how the system works along the time. A worker checks its efficiency at time `t1`. Later, it checks again its efficiency at time `t2`. The time interval taken by the efficiency-guided strategy is always the accumulated working time and accumulated total time since the query started.

We compare the results obtained with the efficiency-guided strategy with the results obtained with the work-guided strategy. Our objective is to evaluate if the efficiency-guided strategy performs better than our target performance for all benchmarks. As before, all programs were started with a single team of workers. Each program was run 10 times. We ignored the best, worst and some inconsistent times and averaged the remaining times to produce speedups. The results for the work-guided strategy were taken from the speedup figures shown in table 1.

Bms	Target/Cfg		Reconfig	
			WGS	EGS
ndetdet	8.487	*(10M)*	9.831	9.219
mixed	3.997	*(6M4S)*	4.004	3.952
detndet	3.012	*(8M2S)*	3.157	3.686
bqu8	7.516	*(10M)*	4.474	7.405
chat	7.334	*(10M)*	7.034	7.046
cypher	5.922	*(10M)*	6.116	5.813
mutest	5.959	*(10M)*	6.025	6.236
bqu6	3.950	*(9M1S)*	2.920	3.071
flypan2	6.203	*(9S)*	6.307	5.285
bt_cluster	6.002	*(9S)*	5.793	5.667
flypan4	5.079	*(9S)*	4.961	4.479
scanner	5.485	*(2M8S)*	5.414	4.249
road_markings	4.887	*(9S)*	4.269	5.192
bcnet	3.746	*(2M8S)*	3.942	3.430
cross11	2.666	*(9S)*	2.917	2.836
flypan5	2.862	*(2M8S)*	3.382	3.128
cross6	2.529	*(9S)*	2.260	2.124
H.mean	4.407		4.258	4.230

Table 2: EGS x WGS x TARGET, AT 10 WORKERS

Table 2 shows the overall performance of the efficiency-guided strategy (column `EGS`) compared with the work-guided strategy (column `WGS`) and with the best fixed configuration (column `Target/Cfg`). Results were taken at 10 processors. As before, the best fixed configuration is shown between parentheses. The last row of the table (`H.mean`) shows the overall performance of the strategies and of the target performance expressed in terms of the harmonic mean.

The efficiency-guided strategy has performance very close to the work-guided strategy, and the overall performance, given by the last row of table 2, is very similar to the overall performance of the work-guided strategy.

We showed before that the work-guided strategy performed much better than our target performance, because the overall speedups achieved by the work-guided strategy were comparable with the overall speedups achieved with the target performance. But looking at individual benchmarks, we observed that the work-guided strategy was not achieving our target performance for some cases. The efficiency-guided strategy, although presents an overall result that is comparable with the overall performance of the work-guided strategy, performs better for each benchmark individually. For the program `bqu8`, whose speedup was very low compared with the speedup achieved by the best fixed configuration, the efficiency-guided strategy performs much closer to the best fixed configuration than the work-guided strategy. For some other computations, although the speedups drop with relation to the best fixed configuration, the efficiency-guided strategy still performs quite well.

In summary, we conclude that the efficiency-guided strategy uses a better heuristic than the work-guided strategy, by achieving similar or better results than the work-guided strategy overall and for each individual benchmark.

Although the work-guided strategy has advantages and shows the benefits of having a reconfigurer in Andorra-I, it has a disadvantage when dealing with benchmarks that contain predominantly or-parallelism and a very small amount of and-parallelism that arises from the or-parallel branches. As showed before, the work-guided strategy does not reach the target performance for this particular kind of problem, typified by the benchmark `bqu8`.

The main reasons for this phenomenon is that the computation for `bqu8` has a non-determinate phase that is much bigger than the determinate phase. And the non-determinate phase has very fine grained and-parallelism. Because each choicepoint created has only two alternatives, and the run queues of goals have plenty of non-determinate goals, the work-guided strategy fails to change some slaves into masters. This happens mainly because the work-guided strategy uses very crude estimates of sizes of work to distribute processors among and-work and or-work.

Besides being less short-sighted, the efficiency-guided strategy brings another advantage that is to give a solution to the race problem existing when several workers want to be redeployed to the same source of work. This is controlled because workers are subject to different low cost redeployments at each time of the computation instead of always trying the same low cost redeployment.

6 Conclusions

In this work we performed two important evaluations. First, we evaluated the benefits of using *dynamic* reconfiguring in the Andorra-I system. Second, we studied the performance of two different reconfiguring strategies to distribute and-work and or-work in Andorra-I, the *work-guided* strategy and the *efficiency-guided* strategy.

We showed how Andorra-I with dynamic reconfiguration compares with Andorra-I with no dynamic reconfiguration using the work-guided reconfigurer as an example of dynamic reconfiguring. We did a comparison for some plausible fixed configurations, where we give different weights to and-parallelism and to or-parallelism. We concluded that Andorra-I with dynamic reconfiguring is much better than Andorra-I without a reconfigurer, for three reasons: (1) The user does not have the burden of deciding which configuration to use; (2) Overall, the reconfigurer performs much better than any one of the fixed configurations; (3) For individual benchmarks, the reconfigurer performs as well as or better than any of the fixed configurations.

Although dynamic reconfiguring was applied to a particular parallel logic programming system, we believe that the same idea can be applied to other parallel logic programming system that aims to exploit both and- and or-parallelism.

Acknowledgements

The author is indebted to David H. D. Warren, Vítor Santos Costa, Rong Yang and Tony Beaumont, for their invaluable help and discussions. This research was supported by NUTES/UFRJ and CNPq, under grant 202270/89.0.

References

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

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

[3] Mark Crovella, Prakash Das, Czarek Dubnicki, Thomas LeBlanc, and Evangelos Markatos. Multiprogramming on Multiprocessors. In *Third IEEE Symposium on Parallel and Distributed Processing*, pages 590–597, December 1991.

[4] Inês Dutra. Strategies for Scheduling And- and Or-Work in Parallel Logic Programming Systems. In *Proceedings of the 1994 International Logic Programming Symposium*, pages 289–304. MIT Press, 1994. Also available as technical report CSTR-94-09, from the Department of Computer Science, University of Bristol, England.

[5] Inês Dutra. *Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System.* PhD thesis, University of Bristol, Department of Computer Science, February 1995. Ph.D. thesis.

[6] Gopal Gupta and M. V. Hermenegildo. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *LNCS 569, ICLP'91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, pages 146–158. Springer-Verlag, June 1991.

[7] Gopal Gupta, V. Santos Costa, R. Yang, and M. V. Hermenegildo. IDIOM: Integrating Dependent and-, Independent and-, Or-parallelism. In *Proceedings of the 1991 International Logic Programming Symposium*, pages 152–166. MIT Press, October 1991.

[8] 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.

[9] 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.

[10] Johan Montelius. Penny, A Parallel Implementation of AKL. In *ILPS'94 Post-Conference Workshop in Design and Implementation of Parallel Logic Programming Systems, Ithaca, NY, USA*, November 1994.

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

[12] V. Santos Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I.* PhD thesis, Department of Computer Science, University of Bristol, August 1993.

[13] 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.

[14] David H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues. In *Proceedings of the 1987 International Logic Programming Symposium*, pages 92–102, 1987.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] John Zahorjan and Cathy McCann. Processor Scheduling in Shared Memory Multiprocessors. In *1990 Conference on Measurement and Modelling of Computer Systems*, pages 214–225. ACM Press, May 1990.