Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System

Inês de Castro Dutra

A thesis submitted to the University of Bristol in accordance with the requirements for the degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science.

February 1995

Abstract

Parallelism and logic programming are two fields that have been successfully combined, as is shown by recent implementations of parallel logic programming systems. There are two main sources of parallelism in logic programming, namely or-parallelism and and-parallelism. Or-parallelism is exploited when several alternative clauses to 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-parallelism 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.

In parallel logic programming systems which exploit both and-parallelism and or-parallelism, a problem that arises 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 thesis 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, and propose scheduling strategies that aim at efficiently distributing processors between and-work and or-work.

We study general criteria that every scheduling strategy should meet to reconfigure processors without incurring too high overheads in the Andorra-I system. We propose two different strategies to reconfigure processors between and-work and or-work based on these criteria. One strategy, *work-guided*, guides its decisions by looking at the amount of current and-work and or-work available in an application during execution. The other strategy, *efficiency-guided*, is designed around the intuition that processors should spend most of their execution time doing useful work, i.e., performing reductions.

Our benchmark set consists of programs that contain both and-parallelism and or-parallelism and programs that contain a single kind of parallelism. We compare the results produced by our strategies with the results produced by a version of Andorra-I that provides only a fixed division of processors between and-work and or-work. Results show the benefit of adding a reconfigurer to Andorra-I and show that both strategies perform better than any plausible fixed configuration for all benchmarks. Although our work is focussed on the Andorra-I system, we believe that both strategies could be applied to other parallel logic programming systems that aim at exploiting both and-parallelism and or-parallelism.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or institution of learning.

Inês de Castro Dutra

Acknowledgements

Several people have contributed to the realisation of this work and for my (in) formation in the logic programming field.

I would like to express my gratitude to my advisor, David H. D. Warren, for providing me with guidance, support and understanding. His insightful comments and suggestions regarding the presentation and content of this dissertation have improved it greatly.

Logic Programming was first introduced to me by my ex-supervisor, Claudio Amorim, while I was doing my M.Sc. course in UFRJ/Brasil. After that I got to know the mysteries of the PLM and WAM abstract machines through Priscila that had all the patience in teaching me everything she arduously got to know facing several difficulties and very hard work. Ricardo, always in a very good humoured mood, was of very much help and support during the time we worked together in the Prolog project.

I am indebted to the members of the Gigalips project, particularly David Warren, Vítor Santos Costa, Rong Yang, Tony Beaumont, Raéd Sindaha, Gopal Gupta and Kish Shen, for their invaluable comments and discussions.

David, my advisor, was of great support during all my stay in Bristol. He was always ready to discuss about any question and encouraged me to carry on this work.

Vítor has been my "guru" in all aspects of computer science. His knowledge of Unix systems and computer science in general made me think that I had to go back to the university and learn everything again properly! I learned very much from him.

Rong, with her kindness "saved" me several times while I was running programs in Andorra-I. Tony was also always friendly and helped me to understand the or-scheduling implementation issues in Aurora and Andorra-I. Raéd gave me great support giving comments about my work and explaining to me many differences among or-parallel schedulers.

Thanks to Kish for many "hot" discussions in several subjects!

Thanks also to all the people that contributed with applications to run in Andorra-I and helped to test and debug the system. Lydia Hluchoweckyj kindly agreed in giving us her Prolog code for experimenting. People from the British Telecom, particularly Dewi Munaf, Brian Tester and Barry Crabtree collaborated very much writing several real applications that helped to stress the usefulness of exploiting both and- and or-parallelism in logic programming systems.

Working in Bristol has been a very enriching experience that broadened my view in the logic programming area. I had the chance of knowing the state of the art in logic programming research by sharing the same roof with very well-known logic programming researchers like John Gallagher, Steve Gregory and John Lloyd, not to mention my own advisor, David Warren. I had the chance to participate of workshops, conferences and meetings. I also had access to several kinds of information like citation indices, internet, newsgroups, www's, mosaic and several other facilities that broadened my view in computer science in general.

I would also like to thank all anonymous people that created the internet and www's which made the world so small allowing me to `talk` to my family and friends back home, and allowing me to browse through bibliography sites, ftp sites, netinfo and more wonders!

I cannot leave out my sincere gratitude to all my friends in Bristol who tolerated my ups and downs (more ups than downs, fortunately!) during my research work.

I would like to thank professors Nilma, Maria Alice, and Eliana Claudia from NUTES/UFRJ, for giving me permission to leave my job and carry out these invaluable years of research, and to professor Claudio Amorim from COPPE/Sistemas to give me the permission to come back to England to finish my work.

Last, but not least, I would like to thank my family for all the support and love they gave me throughout my life.

Thank you very much you all!

This dissertation has benefited from comments of several people. Many thanks to David, Rong, Tony, John, Steve, Vítor, Raéd and William that helped me with several drafts of this work.

This research was supported by CNPq, Brazilian Research Foundation, under grant 202270/89.0. I would like to thank them for providing me with funds to live in Bristol and pay my (huge!) "overseas" fees to study at the Computer Science Department in the University of Bristol.

To Vítor

Contents

List of Figures

List of Tables

List of Programs

List of Algorithms

Chapter 1

Introduction

In an attempt to speed up the execution of programs that are executed by only one processor, new computer architecture models have been designed that connect together several processor units. These models aim at executing parts of a user program in parallel, in order to reduce the total time taken to execute the whole program. The models are classified as *synchronous* or *asynchronous*.

Synchronous models require that each processor operates on the data at the same global time. SIMD (Single Instruction streams over Multiple Data stream) machines, vector/array processors, and systolic processors are examples of synchronous machine architecture models. In the SIMD model, several processors perform the same operation on multiple data at the same time. In vector/array processors, instructions are executed synchronously on data in an assembly line fashion. Systolic processors combine SIMD with vector/array kinds of computation.

Asynchronous computers are classified as MIMD (Multiple Instruction streams over Multiple Data stream) and *reduction machines*. In MIMD models, several processors can perform any kind of instruction in any set of data and the processors proceed independently from each other. In the reduction model, each processor performs reductions in a graph. This model is also called a *data-driven* reduction model. In asynchronous machines, the synchronisation needs to be done via software.

Another taxonomy found in the literature to further classify asynchronous models is the SPMD (Single Programs over Multiple Data) model. In the SPMD model, parallelism is exploited at the data level with one single piece of code executing over multiple data. It is a model similar to SIMD, but with coarser grained parallelism, which means that the size of the tasks to be executed by each processor in a SPMD model is bigger than the size of the tasks to be executed by each processor in a SIMD model. For example, one loop can execute over multiple data of an array by a single processor in a SPMD model.

With respect to memory organization, parallel computers are classified into *distributed memory* systems and *shared memory* systems, although this classification has become blurred with the new memory models that use a global virtual shared memory space associated with high speed local memories. In purely distributed memory systems, each processor has a local view of its own memory, and global data is accessed from other local memories in other processors via physical communication links. Usually these machines do not implement virtual memory, as is the case with Transputer-based systems. In shared memory machines, all processors are connected to a global memory via a common data bus and theoretically can access any data item in memory. Usually, access to a shared data item in a shared memory machine varies from 10 to 10,000 times faster than a remote memory access in a distributed memory machine.

Although MIMD machines with a shared global memory space are relatively new compared with distributed memory machines, they became much more popular than purely distributed memory machines, because they have a much simpler programming model. The shared memory architecture provides a more transparent programming environment to the user, since the user sees the whole address space as only one. In addition, due to very high communication overheads, distributed memory machines are not suitable for running applications that produce very irregular computation patterns. Because of the simplicity of the model, and its suitability for our purposes, our research is conducted in the context of MIMD parallel machines with a global shared memory space associated with caches. Our target is to run *logic programs* in parallel, i.e., to ally the power and expressiveness of logic programs with the power of parallel machines. Still there are a few research groups working with the implementation of parallel logic programming systems for distributed memory machines [14].

Before moving to the motivation and objectives of this work, let us introduce logic programming and logic programs. The logic programming paradigm is based on the idea that programmers should only worry about writing (programming logically) a solution for a problem and ignoring the control flow that will solve the problem. Therefore, a logic program, instead of consisting of statements and control structures, contains only the definition of the problem to be solved, i.e., *rules* and *facts* (*clauses*) that establish relations among objects. Each clause that is a rule is defined by several other relations, that are called *goals*.

One of the advantages of logic programming is that it allows the user to program more *declaratively*, i.e., it leaves the programmer free of the burden of using control structures normally used for imperative programming. Moreover, logic programming has a solid and formal basis in mathematical logic, which is an advantage to the design and utilisation of logic programming languages. If a logic programming language brings the advantage of being declarative and independent of the machine, why not use the same language in a parallel context? Parallelism can be transparently exploited in logic programs because the focus in programming is not the control, but the definition of the problem.

There are several ways that parallelism can be exploited from logic programs. The most common ones are *control-parallelism* and *data-parallelism*. We concentrate on exploitation of control-parallelism, because it is coarser grained than data-parallelism, and therefore easier to parallelise and to obtain reasonable gains in execution time. In control-parallelism we can exploit *and-parallelism* and *or-parallelism*. And-parallelism is exploited when several goals are allowed to execute simultaneously. Or-parallelism is exploited when we allow for several clauses to proceed in parallel.

1.1 Motivation and Objectives

An important problem that remains to be solved in 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 [148], PBA [63], ACE [58], ParAKL [92] and Penny [91], and models of parallel logic programming systems, such as the Extended Andorra Model [140] and IDIOM [62], both and-parallelism 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- and or-work. This is a new and hard problem to be solved in parallel logic programming systems.

The main subject of this thesis 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. 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 at exploiting 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*.

At least three approaches to schedule work in parallel programming systems can be used:

(1) the static approach, where the program can be partitioned at compile-time and each part is allocated to a different processor, provided that we have enough processors to execute all parts;

(2) the semi-dynamic approach, where the runtime system is aided by information generated through compile-time analysis to guide the allocation, and

(3) the dynamic approach, where the scheduler uses some runtime criteria to dynamically allocate processors to tasks.

This thesis discusses scheduling techniques that are based on a dynamic approach. 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.

In order to introduce the subject of scheduling in parallel logic programming systems we give a brief introduction to logic programming and parallelism in logic programming, and show how parallelism can be implicitly exploited in logic programs through the Basic Andorra Model. We concentrate our description on a subset of first-order logic, Horn clauses, that is the simplest basis for most of the logic programming concepts studied nowadays.

1.2 Logic Programming, Parallelism and Andorra

In this section we explain several forms of exploiting parallelism from logic programs and give a brief description of a parallel logic programming model, the Basic Andorra Model.

There are quite a few logic languages available in academic and commercial environments. **Prolog** was the first logic language implemented and it is the most widely used. Prolog was created in order to solve symbolic problems in natural language processing and as an attempt of automating theorem proving processes. Its first implementation dates from 1972, and it consisted of a theorem prover based on linear resolution [100] to solve first order logic problems. The popularity of the language till then was restricted to few groups. The computer science community did not adopt Prolog as a programming language because of its inefficiency. It was not until David H. D. Warren designed the first compiler for Prolog [135, 136] that the language gained more popularity. Currently there are a number of efficient Prolog systems in the market and a wide range of applications written and commercially exploited running on different platforms. Examples include SICStus Prolog [17], Quintus, Mac-Prolog, BIM-Prolog, APL and Arity-Prolog.

In order to surpass the efficiency already achieved in sequential systems, researchers went further and started to study how to exploit parallelism from logic programs, given that the declarativeness inherent in Prolog would make it easy to parallelise. This was a way of improving greatly the speed of the existing systems.

There are two ways of exploiting parallelism in parallel programming systems. The first one is to use control constructs to guide the exploitation of parallelism. By doing that, one is designing a sequential programming language with extra features for control. Parallelism in this case is *explicit*. The second way is to design the language and leave the exploitation of parallelism to the execution system. Parallelism in this case is *implicit*. The same principle applies to parallel *logic* programming languages. We can extract parallelism implicitly from Prolog programs by allowing more than one clause to be tried at the same time, thus giving rise to *or-parallelism*. We can also extract parallelism from Prolog programs by allowing more than one goal to be selected at the same time, thus giving rise to *and-parallelism*. And-parallelism is further classified into *Dependent And-Parallelism* (DAP) and *Independent And-Parallelism* (IAP). Dependent and-parallelism arises when we allow simultaneous execution of goals that share variables. A particular case of DAP is to execute parallel goals that match at most one clause in the program, i.e., there is no nondeterminism under and-goals. Independent and-parallelism only allows simultaneous execution of goals that do not share variables. Usually, systems that exploit any kind of and-parallelism implicitly need preprocessing of programs in order to determine which variables are dependent and which are independent.

In order to exploit full parallelism from a parallel logic programming system, one should allow exploitation of and-parallelism and or-parallelism. There are several ways of combining and-parallelism with or-parallelism:

- we can combine or-parallelism with *determinate* and-parallelism, as in the Basic Andorra Model [139], and Andorra-I [102, 148];
- we can combine or-parallelism with IAP, as in Conery's And/Or process model [27], the ACE model [58] or PBA [63];

- we can combine or-parallelism with DAP as proposed in Prometheus [110];
- a further way of combining both kinds of parallelism is to allow or-parallelism plus IAP, and a restricted kind of DAP, deterministic DAP. The IDIOM [62] model proposes this combination.
- we can allow all possible kinds of parallelism, DAP, IAP and or-parallelism, as proposed in the Extended Andorra Model [140] and AKL [92].

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 [139].

Choosing deterministic goals to execute first is a very interesting way of exploiting and-parallelism in logic programs and it was implemented for the first time in the Andorra-I system (based on the Basic Andorra Model) and in the PNU Prolog system [97]. The system we study in this thesis is the Andorra-I system, that combines and- with or-parallelism, and first executes goals that match at most one clause in the database, i.e. the *determinate* goals. Because of eager execution of determinate goals, Andorra-I can reduce the search space in relation to Prolog, since unnecessary choicepoints are eliminated.

In order to detect which goals are determinate, Andorra-I uses preprocessing [101]. However this preprocessing is less demanding than the preprocessing needed in systems that exploit non-determinate dependent and independent parallelism, because one can detect several conditions for determinacy by doing only local analysis of clauses.

Andorra-I exploits or-parallelism much in the same way as in the Aurora [88] or-parallel system. The bindings for the conditional variables are stored in binding arrays according to the SRI model [138]. It exploits and-parallelism similarly to JAM Parlog [32].

In Andorra-I workers are arranged into teams of workers 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. Thus execution is divided into two distinct phases: determinate and non-determinate. During the determinate phase processing elements in a team (masters and slaves) are cooperating to execute and-parallel work, and during non-determinate phase the master of a team creates new choicepoints to prepare for a new determinate phase.

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 [101]; an engine [148], responsible for the execution of the Andorra-I programs [145, 146, 103], two schedulers, the and-scheduler and the or-scheduler [7], and a *reconfigurer*.

In the next section we discuss and-/or-scheduling issues in the Andorra-I system.

1.3 The Scheduling Problem in Andorra-I

The main objective of the Andorra-I schedulers is the selection of which piece of work to execute next, since we might 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- and or-work are available?

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 is a problem different from selecting which piece of work to execute next. It is a more general problem of *what kind* of work to choose for each processor.

Once this problem is solved, each of the other scheduler components (the and-scheduler and the or-scheduler) can be responsible for finding one of the and-tasks or one of the or-tasks. If many alternatives are left to execute, one of them needs to be selected. This is done by an or-scheduler. If we have many parallel and-goals we also need to select the next to be executed. The and-scheduler is responsible for selecting an and-task, which corresponds to a goal in the run queue (list of goals not yet executed) of a worker.

This thesis 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 reconfigurer, besides adding an automatic way of selecting workers to execute the programs, is responsible for maintaining the execution tree balanced with respect to the number of workers cooperating to exploit and-work and or-work. If the current computation in a branch is very rich in and-parallelism, some idle workers should be redeployed to the rich branch. On the other hand, if there are too many alternatives left in the execution tree, some of the idle workers should take some of these alternatives and start a new branch in the tree.

The general problem of reconfiguring can be summarised through a simple example. The picture shown in Fig. 1.1 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).

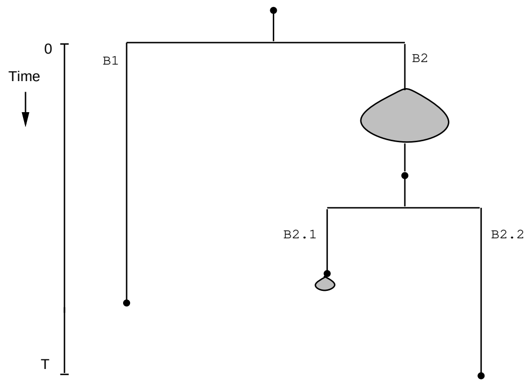


Figure 1.1: AN ANDORRA-TREE

In Figure 1.1, at the root node, there is a choicepoint with two alternatives. The first branch (`B1`) has little or no and-parallelism while the second branch (`B2`) has a reasonable amount of and-parallel work followed by some or-parallel work (represented by branches `B2.1` and `B2.2`). 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`, in figure 1.1.

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.

If possible we would like also to obtain reasonable speedups close to the best achievable.

As was mentioned before, there are at least three approaches to achieve a solution for the scheduling problem: (1) the first one is to use a completely static method, i.e., to feed the parallel system a partitioned Prolog program and get an approximate static schedule; (2) the second approach is to use a semi-dynamic method where the scheduler would at some point make decisions based on information generated at compile-time, and (3) the third method is to devise a good dynamic scheduling strategy that would react to the dynamics of the execution system. With the semi-dynamic approach we need to solve two problems at once: static plus dynamic scheduling.

The first approach is impractical because of the nature of the Prolog applications. Each Prolog program generates a different execution tree and it is a very hard task to statically partition the program to be scheduled. Researchers in Japan [117, 122] use the idea of statically partitioning programs, but applied to committed choice-style programs where the parallelism is exploited with the aid of the control structures. It is also more advisable to use static scheduling in distributed memory machines, because of the communication costs involved in tasks switching from one processor to another. In Prolog, the tasks of partitioning programs or performing some kind of compile time granularity analysis are very difficult, because of (a) the high variance of the input data size, (b) the shape of the trees generated by different queries for the same program, and (c) the difficulty in transforming some recursive predicates into well-formed graphs.

This thesis contributes towards the second and third approaches by trying to solve the dynamic aspect of scheduling.

1.4 Thesis Outline

In chapter 2 we give an overview of logic programming, its history, and the Prolog language. We also discuss the relation between logic and logic programming showing how logic programming inherits its formal basis from mathematical logic.

In chapter 3 we discuss issues related to introducing parallelism in logic programming. We describe the main parallel logic programming systems existing in the literature, and discuss several ways of exploiting parallelism in logic programming: control- and data-parallelism, combination of and- and or-parallelism, unification-parallelism etc.

In chapter 4 we present and discuss scheduling techniques used by some of the parallel logic programming systems found in the literature, and described in chapter 3. We present briefly the Andorra model and discuss recent achievements in scheduling in parallel logic programming systems.

In chapter 5, the Andorra model is presented and the Andorra-I system is described by means of its data structures, master and slave life cycles, synchronisation algorithms and determinacy analysis. The Andorra-I scheduling components: the and-scheduler and the or-scheduler, and the engine are described in more detail.

In chapter 6, we present the motivation for flexibly scheduling and-work and or-work and present two possible solutions, the *work-guided* strategy, and the *efficiency-guided* strategy. We discuss several aspects of reconfiguring workers into teams, and show several problems and possible solutions to deal with them, stressing the need to reduce the parallel overheads of the system in order to obtain good speedups. We show how Andorra-I was modified to support the reconfigurer. In this chapter we also evaluate closely the Andorra-I performance with respect to its main operations. We draw a model of overheads for the system and show how the processors spend their time in different activities of the system.

In chapter 7, the work-guided strategy is described and implementation issues are discussed. Different criteria to decide between and-work and or-work are discussed.

Chapter 8 presents the benchmark set used to test the strategies, and stresses the benefits of using a reconfigurer in Andorra-I through results produced with the work-guided strategy. We also evaluate the work-guided strategy for several benchmarks with respect to a *target performance*, which is defined as being the best performance achieved with a fixed configuration that varies for each benchmark at each number of processors.

In chapter 9 we describe another reconfiguring strategy for Andorra-I, the efficiency-guided strategy. This strategy aims to reduce the overheads of the system and better reconfigure workers in order to exploit more parallelism.

In chapter 10 we compare results obtained with the efficiency-guided strategy with the results shown in chapter 8. We show how the efficiency-guided strategy reduces significantly the idle time and the total number of reconfigurations in the system.

Finally, chapter 11 summarises the research, draws some important conclusions, explains the contribution, and discusses future work.

1.5 Summary

This chapter gave an overall introduction to the nature of the work described in this thesis. We presented the problem to be solved, what solution was proposed, and how it contributed to the field.

Our objective is to run *logic programs* in parallel. So the idea is to combine the power of parallel machines with a programming paradigm that has its basis in the *formalism of mathematical logic* and, therefore has a *declarative* meaning. The chapter began by giving the motivation for *parallelism* and also a very brief description of parallel architectures. We then situated our research in the context of *MIMD* machines. We discussed briefly several forms of parallelism that can be exploited in logic programs, and limited our context to control-parallelism: the exploitation of *and-parallelism* and *or-parallelism*.

Next, we gave a brief overview of systems that exploit and-parallelism only, systems that exploit or-parallelism only, and systems that exploit both forms of parallelism. The *Andorra-I system* is our target system. In the context of Andorra-I we introduced a problem that had not yet been tackled in the scope of systems that exploit both and- and or-parallelism: *how to distribute and- and or-work among processors* during execution. We discussed the problem and possible ways of dealing with it: static, semi-dynamic and dynamic approaches. Our research work takes the *dynamic* approach to distribute work among the processors. We presented the *reconfigurer* as the component of Andorra-I that distributes *dynamically* and- and or-work among processors.

Chapter 2

Overview of Logic Programming

In this chapter we give an overview of Logic Programming and describe the terminology used throughout this thesis. We describe Prolog, the most popular logic programming language with the aid of program examples. We also describe the relation between logic and logic programming. This study is fundamental for the understanding of the subjects discussed in the next chapters.

2.1 History

Logic programming has its basis in studies of formal logic mathematics, in particular, Predicate Calculus [89]. The motivation for creating a logic programming language was driven by the interest in solving several problems that were not very easily solvable by the existing imperative programming model.

The main concern of researchers at around 1970 was, from one side, to build theorem provers [79], and from another side, to do parsing and syntactic analysis for natural languages [25].

Prolog was the first practical logic programming language, designed by Alain Colmerauer with its theoretical groundwork laid by Robert Kowalski, in 1974. Prolog is the most widely known representative of logic programming languages. Most other logic programming languages are derivatives of the Prolog computational model.

The pioneering work of Robinson [100] using unification in theorem proving was of capital importance in the development of Prolog. The successive methods studied for theorem proving (e.g., natural deduction and resolution) led to the present inference mechanism of

Prolog: Selective Linear Definite or SLD clause resolution. The word "definite" refers to Horn clauses with exactly one positive literal, whereas Horn clauses may contain entirely negative clauses.

Logic programming has been largely used in many research centres as an alternative to the imperative programming paradigm. Logic programming is suitable for solving problems of various nature and currently is applied to the solution of symbolic problems mainly, such as theorem proving, natural language processing, databases, expert systems and artificial intelligence problems.

Languages as C, Pascal, Modula-2, etc, with their control structures, statements, assignments and side-effects are suitable for heavy numerical and scientific processing as well as the conventional architectures that run these languages. Logic languages are used for mathematical symbolic processing based on theorem proving. Besides being suitable for non-numerical processing, several other factors make logic programming more attractive to programmers:

- Logic programming languages do not have destructive assignment, where a variable in a program can have several values each time a value is assigned to it. A variable in logic programming can assume one single value during the whole execution. For the user, this prevents unwanted side-effects that could eventually happen during the execution of an imperative program. Logic programming languages use the concept of logical variables that can be assigned just once and that are implicitly quantified. For example, consider the program in figure 2.1.

```
p(X,Y) :- X = 2, X = 3.
```

Program 2.1: THE LOGICAL VARIABLE.

  If variable `X` is non-instantiated, the first goal will *bind* the variable to the value `2`. Later when a goal tries to bind the variable `X` to `3`, the result will be a failure, because the value for `X` cannot be modified. Instead, the variable is consulted (*dereferenced*) and its value `2` is compared with the value `3`. As they are different, a failure occurs.

- A program can produce many solutions for the same question. This means that logic languages automatically allow non-determinism. For example, assume we have the following program, as in figure 2.2. We can ask to this program all pairs such that the relation `likes` is true: `?- likes(X,Y)`. This will produce 3 solutions to the program, basically all pairs of objects expressed by the relation `likes`: `(john, mary)`, `(mary, andrew)`, and `(john, william)`.

```
likes(john,mary).
likes(mary,andrew).
likes(john,william).
```

Program 2.2: NON-DETERMINISM IN LOGIC PROGRAMMING.

- A question can be made with any number of arguments instantiated, what provides more declarativeness and flexibility to the programmer. For example, in program 2.2, we can enquire things such as: `likes(john,X)`, meaning a question about all people that are liked by John; `likes(john,mary)`, that enquires if it is true that John likes Mary. Each of these questions represent a different formula to be proved.

Because the programmer can concentrate more on **what** to solve and not **how** to solve a problem, logic programming languages are called *declarative* languages.

Several research groups have been concentrating on finding other more powerful logic languages such as Gödel [70], Escher [87], Multi-Prolog [12], languages based on equation relational sets [77], languages based on temporal and modal logics, concurrent constraint-based languages [104], higher order languages, and so forth. However Prolog is the simplest and most efficient logic programming language implemented.

In the next section we give an introduction to the Prolog language, and show some program examples.

2.2 Prolog

Prolog is a computer programming language [24, 115] that is used for solving problems that involve *objects* and *relationships* between objects. We use Prolog when we wish the computer to solve problems that can be expressed in the form of objects and their relationships. For example, when we say "Mary likes John", we are declaring that a relationship exists between one object "Mary" and other object "John". Furthermore, the relationship has a specific order: Mary likes John, and this does not imply that John also likes Mary. When we ask: "Does Mary like John?", we are trying to find out about a relationship. In Prolog, this is a *query* and the sentence "Mary likes John" is a *fact*.

As well as using facts, we can also use *rules* to define relationships between objects. For example, "Two people are brothers **if** they are both male and have the same parents" is a rule that tells us *what* it means to be brothers. The rule also tells us *how* to find out if two people are brothers: find if they are male and if they have the same parent. Thus a logic program can have a *declarative* meaning ("what") and a *procedural* meaning ("how"). The declarative meaning reads the clauses statically, textually and understands the definitions corresponding to the rules and facts. The procedural meaning tries to understand **how** the logic program produces a solution.

A procedural meaning of a logic program is obtained by matching the query with the head of a clause with the same name and number of arguments. If the clause is a rule, then we need to solve all goals in the body of the clause by matching each goal with a clause in the program, until we get to the facts and finish execution with success (producing a solution) or until a failure occurs, in which case another solution for the goal that failed is tried. The process of matching used to find a candidate clause uses a *unification* algorithm that applies substitutions to the arguments of the goal. In order to execute a logic program, one therefore needs to use a *goal selection rule*, the rule that will select which goal to execute next. As a goal can have several alternatives, one also needs a *search rule*, that will tell which alternative to select next. The execution of a logic program, as such, produces a *search tree*. The search tree varies according to the goal rule selection, while the search rule does not modify the search tree.

Prolog implements SLD-resolution [100, 71] and employs a left-to-right goal rule selection and depth-first search strategy. This means that Prolog always starts to execute the goals in a clause from the leftmost to the rightmost goal. It solves a goal by always exhausting one alternative before trying the next one. A resolution step in Prolog is also called a *reduction* or a *logical inference*.

As compared with imperative languages, Prolog has no typing system, no control structures and no destructive assignment. Its philosophy is different from the imperative languages because it does not try to solve the problems assuming a language based on the Von Neumman architecture. In other words, the programmer does not need to know that the machine has memory, control unit and ALU. Problems can be defined in a logical manner without knowing about the structure and execution method of the underlying computer architecture.

In this section we will discuss features of the Prolog language such as: facts, rules, queries, conjunctions, variables, lists, recursion, forward and backward execution.

In order to define a fact in Prolog we use the following syntax (Edinburgh syntax [135]):

```
likes(mary,john).
```

where the names of all relationships and objects must begin with a lower-case letter. The relationship is written first, and the objects are written separated by commas, and enclosed by a pair of round brackets. The full stop character '.' must come at the end of a fact.

In order to define a rule, we use the following syntax:

```
brothers(X,Y) :- male(X), male(Y), parent(Z,X), parent(Z,Y).
```

This rule states that two people (`X` and `Y`) are brothers if they are male, and have the same parent (`Z`), where `X`, `Y`, and `Z` are variables.

Here is another example of a rule:

```
bird(X) :- hasfeathers(X), hasbeak(X)
```

This is a rule that says that if X has feathers and X has a beak, then X is a bird.

A fact is a tautology in a logic program. For example, `human(socrates)` says that it is always true that Socrates is a human. Rules and facts in a program can also be called *clauses*. The names given to rules (e.g., `bird`), or to facts (e.g., `human`) are also called predicates. We can have several clauses to the same predicate. The left hand side of a rule is called the *head* of the clause. The right hand side of a rule is called the *body* of the rule (or of the clause) and the body is a collection of 0 or more *goals*. For example, `hasfeathers(X), hasbeak(X)` corresponds to the body of the clause `bird`, where `hasfeathers(X)` and `hasbeak(X)` are two goals. A clause with no goals is a fact. Usually we refer to a predicate by its name (e.g., `bird`) and its *arity*, that is the number of *arguments* the predicate has. In that case, `bird/1` is a predicate that has one argument.

When executing a program, the Prolog system exhaustively uses a *top-down recursive* algorithm to the forward execution, and a *backtracking* algorithm for the backward execution. It creates a *choicepoint* for every call to a goal that has more than one candidate clause.

In order to prune the search space and avoid explosion of the search, Prolog implements the *cut* operator, syntactically represented by the symbol '!'. Program 2.3 shows an example of the utilisation of the cut operator (the program uses the Edinburgh syntax for Prolog). The predicate `p/2` is defined with four clauses. The second clause says that if `s/2` has a solution, then all remaining clauses for `p/2` (after the second clause) must be pruned. Furthermore if `s/2` has more than one solution and `t/2` fails, no other solution for `s/2` will be tried. Some other logic languages use another pruning operator called *commit*, represented syntactically by the symbol '|'. The commit operator is also used to control search explosion, but instead of pruning only clauses after the one that contains the commit (as the cut operator does), it also prunes all the clauses before the clause that contains the commit operator. In fact, this is an operator used in concurrent logic languages. Notice that this operator has no meaning in Prolog, since the selection of clauses in Prolog obeys the order of the program text. We will talk about it later in the next chapter, when we discuss about parallel and concurrent logic systems.

```
p(X,Y) :- q(X,Y), r(X,Y).
p(X,Y) :- s(X,Y), !, t(X,Y).
p(X,Y) :- ....
p(X,Y) :- ....
```

Program 2.3: USE OF THE CUT OPERATOR.

Clauses after the clause that contains the cut operator, or clauses before and after the clause that contains the commit operator, are said to be *speculative* work, because we do not know before we reach the cut or commit operators whether they are going to be executed during the normal sequential Prolog execution.

2.3 Program Examples

Program 2.4 shows an elegant Prolog definition of the predicate `length` that defines the number of elements of a set. The syntax used is the Edinburgh Prolog syntax [135]. The first clause simply deals with the basic case of the definition that is: "the size of an empty set is zero", where the empty set is represented by the syntactic notation "`[]`". In case the set is not empty (represented in Prolog as `[H|T]`, i.e., a set that contains at least one element), the definition follows beautifully: take one element out of the full set obtaining a new set `T`, the size of the full set will be the size of `T` + 1!

This is the declarative meaning of this program. And in fact we do not need to know how the Prolog system solves this problem. We know that the definition is correct.

```
length([],0).
length([H|T],N) :- length(T,N1), plus(N,N1,1).
```

Program 2.4: LENGTH OF A LIST.

The operational semantics of this program involves understanding the Prolog machinery to know **how** the program is going to produce an answer. Given a query `?- length([a,b,c],N)`, the system will try recursively to solve the predicate `length`, always removing one element of the list (i.e., breaking the big list in small lists: using an algorithmic decomposition technique) until it gets to the empty set, that is to the basic case when `N` is *instantiated* for the first time to the value `0`. For all levels of the recursion, `N` will now be incremented by `1`, producing the solution `N = 3` at the top level of the recursion (the query level). Note that `N` is a logical variable that assumes several instances for each level of the recursion. Internally, a different memory location exists for each `N` at each recursion level. Notice that in this program we use two basic features of Prolog programs: *recursion* and *lists*.

In addition, when we defined length of a list L as the length of a list L1 (without the first element) plus 1, we are not concerned about **how** the length of any list is going to be computed. We give a recursive definition of the length of a list. This is pure mathematical logical reasoning. This predicate has a very logical definition. One could think of implementing such a **recursive** algorithm in an imperative language, but it is much more efficient and natural for a computer programmer to think, instead of a recursion, of an **iteration** to repeat the operation of **counting** the elements until the list is empty, a data structure for **storing** the list of elements, and an **accumulator** to store the partial length of the list.

This simple example gives a flavour of the power of the language.

Logic programs written as Horn clauses cannot always be used to solve any kind of problem. There are cases where programs are infinite and do not produce any solution, depending on the order we arrange clauses and goals. For example, assume examples in program 2.5 and program 2.6, that define the same family relation ancestor, but with goals arranged in different order.

The first clause of this program states that every parent is an ancestor. The second clause

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z),ancestor(Z,Y).
```

Program 2.5: ANCESTOR (1): LOGIC PROGRAM EXAMPLE WRITTEN IN PROLOG.

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),parent(Z,Y).
```

Program 2.6: ANCESTOR (2): LOGIC PROGRAM EXAMPLE WRITTEN IN PROLOG.

states that the relation ancestor is recursive, and in this case, therefore, can be infinite.

As long as we have a finite database for the predicate *parent*, if we exchange the first clause with the second one, in program 2.5, this program executes correctly and does not loop. On the other hand, if we exchange the two clauses of program 2.6, this program will never stop and produce a solution, although its declarative semantic is logically correct. If we run the program 2.6 as it is, without changing the order of the clauses, and try to find all solutions, one solution will be found, but the program eventually will loop.

The program `ancestor` illustrates that even programs that are defined correctly may not work with any computation rule. Although logic programs that have pure declarative meaning can be stated by clauses written in any order, or goals written in any order, programs written in Prolog do not have this property. Because of the selection mechanism of Prolog where goals are executed in left to right order and clauses are selected from the top to the bottom of the program text, programmers need to be aware of the order.

The following program illustrates again how the Prolog execution mechanism can change the behaviour of a program when we change the order of the clauses. The program shown in Figure 2.7 defines deleting the first element of a list that matches with a given element. The first clause states that if the element `X` is found (represented as the head of the list that is the second argument of the predicate `delete`), the predicate should return the list `Y` without the element `X`. The second clause recursively searches for the element `X` in the list

`[Y|Z]`, where `X` is supposed to be different from `Y`. Because of this last subtle condition (`X` needs to be different from `Y`), if we exchange the first clause with the second clause, the predicate takes a slightly different meaning: instead of deleting the first element `X` from the list, it will delete the last element. So if we have a query: `?-delete(a, [a,b,a,a],D)`, the result will be the list `[a,b,a]`, instead of `[b,a,a]`, i.e., it will delete the last occurrence of `X` from the list.

```
delete(X,[X|Y],Y).
delete(X,[Y|Z],[Y|W]) :- delete(X,Z,W).
```

Program 2.7: DELETE ELEMENT FROM A LIST.

In most Prolog programs, order is not very important if we define the predicates in a correct and complete form, e.g., by introducing `not(X=Y)` to the `delete` program. Still most programmers rely on clause ordering, and sometimes, goal ordering, in order to have more efficient programs.

When we move to a parallel environment to run logic programs, the problem of ordering goals and clauses is even more complex if we want to keep the same semantics of Prolog, and be able to port Prolog programs and speed them up.

2.4 Prolog and Logic

If we wish to discuss how Prolog is related to Logic, and this discussion is very important, because it establishes the formal basis to the language, we must first of all establish what we mean by Logic. Logic was originally devised as a way of representing the form of arguments, so that it would be possible to check in a formal way whether or not they were valid. We can use logic to express propositions, the relations between propositions and how one can validly *infer* some propositions from others. The particular form of Logic that we talk about is the *Predicate Calculus*. Predicate Calculus is a formal language whose essential purpose is to symbolise logical arguments in mathematics. The sentences in this language are called *well-formed formulas* (wffs). By interpreting the symbols in a wff we obtain a statement which is either true or false. In Predicate Calculus we define operators, variables, functions, predicates, and inference rules. We represent objects by *terms*, and a term is of one of the following forms:

- A constant symbol. This is a symbol that stands for a single individual concept. We can think of this as being a Prolog atom as `mary` or `john`, or `likes`.
- A variable symbol. This is a symbol that we may want to stand for different individuals at different times. We can think of them as Prolog variables and we use the Prolog syntax. Thus `X`, `Y` and `Woman` are variable symbols.
- A compound term. A compound term consists of a *function symbol*, together with an ordered set of terms as its *arguments*.

Thus in Predicate Calculus the ways of representing objects are the same as in Prolog.

Predicate Calculus encompasses several classes of wffs:

1. Propositional Calculus

   a) Propositional constants p_i^0

2. Quantified Propositional Calculus

   a) Propositional constants p_i^0

   b) Propositional variables P_i^0

3. Equality Calculus

   a) Individual constants a_i

   b) Individual variables x_i

4. First-order predicate calculus (with equality)

   a) n-ary ($n \geq 0$) function constants f_i^n

   b) n-ary ($n \geq 0$) predicate constants p_i^n

   c) Individual variables x_i

5. Second-order predicate calculus (with equality)

   a) n-ary ($n \geq 0$) function variables F_i^n

   b) n-ary ($n \geq 0$) predicate variables P_i^n

   c) n-ary ($n \geq 0$) function constants f_i^n

   d) n-ary ($n \geq 0$) predicate constants p_i^n

   e) Individual variables x_i

Not all of these classes have an algorithm that computes the validity of any formula/proposition.

Considering that we are interested in the class of **valid** wffs, we look for an *algorithm* that takes any wff of the predicate calculus as input and determine whether or not the wff is valid (this is known as the validity problem of the predicate calculus). Gödel found that the validity problem of the (second-order) predicate calculus is not partially solvable, that is, there is no algorithm (Turing machine) that takes any second-order wff as input and reaches an ACCEPT halt if the wff is valid, but may reach a REJECT halt or loop forever if the wff is nonvalid. Gödel also proved that there is no complete deduction system for the second-order predicate calculus, but there are complete deduction systems for the first-order predicate calculus [89].

From these findings by Gödel, the logic programming practitioners concentrated their efforts on discovering algorithms for systems based on the first-order logic. The work of the logic programming researchers started when the *resolution* algorithm was proposed by Robinson [100].

Resolution is a single-rule refutability technique for logical formulas that, at least in principle, is suitable for using on the computer. Resolution can be applied to some classes of logic mentioned before: propositional logic, first-order logic, and the so-called Horn-fragment of first-order logic, in the context of which is called (definite) *logic programming*. Resolution in propositional logic forms the basis for the first-order version. Though the version for the Horn-fragment appears to be just a special case of the one for full first-order logic, it nevertheless is the most important of the three since it can be given a computational component. As a recent logic programming symposium puts it, "initial interest in logic programming derives from the discoveries that a simple fragment of logic, Horn clauses, could be given a computational interpretation, and that a programming language, Prolog, could be designed from that simple fragment" [41].

The resolution procedure takes as argument a set of expressions in a simplified version of predicate calculus, called *clausal form*. The symbols, terms, and atomic sentences of clausal form use the same as those in ordinary predicate calculus. Instead of logical and quantified sentences, however, clausal form has literals and clauses.

In order to express propositions/clauses about objects we must be able to express relationships between objects. We do this with *predicate symbols*. An *atomic proposition* consists of a predicate symbol, together with an ordered set of terms as its arguments. This is the kind of thing that can appear as a Prolog goal. Out of atomic propositions we can make compound propositions in various ways. There are several ways in which we can make more complicated propositions out of simpler ones. First, we can use *logical connectives* such as “not”, “and”, “or”, “implies” and “is equivalent to”. Table 2.1 presents possible propositions, where α and β are any propositions.

Connective	Predicate Calculus syntax	Meaning
Negation	$\neg\alpha$	“not α”
Conjunction	$\alpha \wedge \beta$	“α andβ”
Disjunction	$\alpha \vee \beta$	“α or β”
Implication	$\alpha \rightarrow \beta$	“α implies β”
Equivalence	$\alpha \equiv \beta$	“α is equivalent to β”

Table 2.1: PREDICATE CALCULUS CONNECTIVES

When introducing variables in a proposition we use *quantifiers*, which provide a means of talking about sets of individuals and what is true of them. Predicate Calculus provides two quantifiers. If v represents any variable and P any proposition, we can summarise them as in table 2.2.

Quantifier	Predicate Calculus syntax	Meaning
$\forall$	$\forall v, P$	P is true whatever v stands for
$\exists$	$\exists v, P$	there is something that v can stand for such that P is true

Table 2.2: PREDICATE CALCULUS QUANTIFIERS

The first quantifier is called the *universal quantifier* because it talks about everything in the Universe. The second is called the *existential quantifier*, because it talks about the existence of some object(s). As examples of uses of quantifiers, we can have, for instance:

$$\forall X, man(X) \rightarrow human(X)$$

meaning that whatever `X` we choose, if `X` is a man then `X` is human. Other example with the existential quantifier:

$$\exists Z, father(john, Z) \wedge male(Z)$$

meaning that Z can assume some value such as John is the father of `Z` and `Z` is male. In English, this stands for simply: *John has a son*.

Predicate Calculus formulae expressed in terms of implication and equivalence can be rewritten in terms of *conjunction*, *disjunction* and *negation*. In fact there are many more identities of this form, and the expressive power of the propositions would not be reduced if we do not use quantifiers and implications, for instance. As a result of the redundancy, there are many ways of writing down the same proposition. If we wish to carry out formal manipulations on Predicate Calculus formulae, this turns out to be inconvenient. Therefore we will consider how a Predicate Calculus proposition can be transformed in a *clausal form*, where there are fewer ways of saying the same thing. In fact, it will turn out that a Predicate Calculus proposition in clausal form is very much like a set of Prolog clauses. The conversion of a Predicate Calculus formula into clausal form has six main stages:

- Removing Implications. At this stage we remove all implications of the kind a $\rightarrow$ b by its equivalent formula $\neg$ a $\vee$ b.
- Moving negation inwards. At this stage we apply negation to all components of a compound formula.
- Skolemising. This stage is when we remove all existential quantifiers. This is done by introducing new constant symbols, *Skolem constants*, in the place of the variables introduced by the existential quantifiers.
- Moving universal quantifiers outwards and making them implicit
- Distributing $\wedge$s over $\vee$s
- Putting into clauses

As an example of transforming a proposition in a clausal form, we can have, for example:

$$\forall X, (\forall Y, (person(Y) \rightarrow respect(Y, X)) \rightarrow king(X))$$

This proposition means that, if everybody respects a person then that person is a king. When we remove the implications we get:

$$\forall X, \neg(\forall Y, (\neg person(Y) \vee respects(Y, X)) \vee king(X))$$

Moving negation inwards:

$$\forall X, \exists Y, (person(Y) \wedge \neg respects(Y, X)) \vee king(X)$$

Skolemising, where *f* is a Skolem function.

$$\forall X, (person(f(X)) \wedge (\neg respects(f(X), X) \vee king(X))$$

Removing universal quantifiers (they are implicit):

$$(person(f(X)) \wedge (\neg respects(f(X), X) \vee king(X))$$

We now go to stage 5 where conjunctions do not appear within disjunctions:

$$(person(f(X)) \vee king(X)) \wedge (\neg respects(f(X), X) \vee king(X))$$

This amounts to two clauses:

- $person(f(X)) \vee king(X)$, and
- $\neg respects(f(X), X) \vee king(X)$

In Prolog we write these clauses using a different notation. In Prolog the convention to write propositions in clausal form is to write the unnegated literals first and without the $\neg$ symbol, and the negated ones second. The two groups will be separated by the sign ':-' (it was common sense that this was a nice way of representing the implication symbol as a ASCII Prolog program symbol). In this notation, a clause with negated literals k, l, ... and unnegated literals a, b, ... would come out as:

```
a; b; ... :- k, l, ... .
```

Following this convention, the clause shown before comes out as:

```
king(X); person(f(X)) :-.
king(X) :- respects(Y,X).
```

Prolog, however, cannot represent more than Horn clauses with at most one positive literal at the left-hand side of the implication. Therefore, although the clausal form above can be written the way it is, Prolog cannot execute it for its computational complexity. There have been several efforts at direct implementation of full logic, but all of them turned out to be grossly inefficient. Improvements led to the resolution method of Robinson over Horn clauses with at most one positive literal at the head. One part of the resolution method has remained unspecified: how to choose between pairs of clauses and the sub-disjunctions of these clauses to be resolved. Substantial effort has been devoted toward developing strategies for making this choice. The most important strategies are described in Nilsson [50] and Chang & Lee [18], although many researchers have been studying different selection rules.

2.5 Discussion

As we conclude from the examples and from its formal logical basis, logic programming can be a very powerful paradigm. More and more applications are written in Prolog nowadays, vide the Prolog 1000 database [94] that contains already more than 500 applications in all areas: databases, natural language processing, planning, genetics, networking, etc. All these contribute to the popularity of logic programming. Logic programming brings together clarity, expressiveness and declarativeness, very important aspects of a programming paradigm. Faster compilers for Prolog have been built [133, 131, 42], and this also contributes to the spreading of the language. Van Roy in a recent article [132] gives a very good overview of the wonder years of sequential Prolog implementations. The most recent compilers use many techniques to improve sequential execution of Prolog. Many compilers use global analysis to optimise code, and use sophisticated mechanisms of indexing in order to reduce the overhead of dereferencing and unification. Some others use information given by the user in the program such as predicate *modes*. These modes inform the compiler when to generate code for unification of subarguments in order to optimise the abstract machine code. Other techniques include register allocation optimisation, choice of structure copying and structure sharing, unfolding of predicates, transformation of Prolog code into C code [121], and program transformation through partial evaluation and abstract interpretation.

prog name	Andorra-I	JAM	Aurora	Muse
nrv400	8.25	8.37	–	–
bt_cluster	9.37	9.70	–	–
bt_wms	3.32	–	–	–
road_markings	6.24	–	–	–
chat_80_db5	7.30	–	7.30	5.91
5x4x3_puzzle	9.66	–	9.51	8.69
warplan	1.20	–	2.63	1.06
protein_all	6.81	–	9.49	8.64
protein_1st	2.78	–	4.10	3.12
fly_pan	6.88	–	–	–
scanner	5.47	–	–	–
cipher	5.65	–	–	–

Table 2.3: SPEEDUPS OF SOME PARALLEL LOGIC PROGRAMMING SYSTEMS AT 10 PROCESSORS

Many projects have been successful in attempting to join *logic programming* with *parallelism*. Either trying new technologies or joining the already known technology employed in sequential systems with parallel implementation techniques. The results obtained so far are quite good and justify the utilisation of parallel logic programming systems as a computing system for end-users. To confirm this, table 2.3 presents speedups of some current parallel logic programming systems for a wide range of applications [147]. The performance is reasonably good, with some of these systems running up to 8 times faster in a parallel machine with 10 processors than sequential implementations as optimised as SICStus Prolog 2.1. The basic performance of these systems is close to the basic performance of sequential Prolog systems.

Because logic programming relies less on the control and more on the logical definition of the problem, it is possible to execute Prolog clauses and goals implicitly in parallel given that we have enough processors to do the job. This is the next step beyond uniprocessor speed limitations.

2.6 Summary

In this chapter, we concentrated on giving an overview of *logic programming*, since this is one of the basis to this research work. We presented the history of logic programming and how it came to be utilised. We then presented *Prolog*, the representative language to program in logic. We showed through several program examples the main characteristics of the language, its syntax and semantics. The relation between Prolog and logic was stressed, as this is a very important subject that supports the advantages of using the logic programming paradigm. Lastly, we discussed the most recent achievements in sequential logic programming systems and show that exploitation of parallelism in logic programming is the immediate step towards having more efficient systems.

Chapter 3

Parallelism and Logic Programming

Knowing that logic programming is a very powerful paradigm capable of solving many important problems in symbolic computation, many researchers have been working on the combination of logic programming with parallelism. This is a very important step towards improving the execution of logic programming systems, and undoubtedly a more powerful paradigm than sequential logic programming. In this chapter we will discuss about how parallelism can be exploited from logic programs. We also discuss the problems and difficulties related to exploiting parallelism in logic programming.

The following discussion is important because the solution to the problem we want to solve, which is how to distribute and- and or-work in logic programming systems, depends on how the parallel logic programming system is implemented, what kinds of parallelism are exploited, what are the sources of overheads and what architecture is being used.

3.1 Main Issues in Adding Parallelism to Logic Programming

3.1.1 What kind of Parallelism to Exploit?

Regarding the programming paradigm, parallelism can be exploited implicitly or explicitly. In order to exploit parallelism explicitly, the programmer is responsible for using language constructs to mark points in the program where the parallelism is exploited. Parallelism is exploited implicitly when the programmer does not need to worry about annotating the programs with constructs to guide the parallelism. To exploit parallelism explicitly or implicitly requires that the compiler generates code that can be executed in parallel. If parallelism is exploited implicitly, the compiler needs to extract characteristics of the program that would lead to exploitation of full parallelism. Usually this is done through preprocessing of programs. These characteristics are usually given by the parallel model or system that runs the program. If parallelism is exploited explicitly, the compiler only needs to generate code for the parallel constructs in order that these constructs can be executed in parallel by the underlying system. A combination of exploiting parallelism implicitly and explicitly is also possible. The programmer can use language constructs to indicate what is to be executed in parallel. In addition, the program can be preprocessed in order to extract possible sources of parallelism that are not annotated by the user.

The advantage of exploiting parallelism implicitly is that we can run any Prolog program originally written to be run on a sequential system without any modifications. Therefore we have a portable system for Prolog programs. This is also true for other parallel programming systems. The disadvantage is that the system might not be able to exploit any parallelism because the program was originally written to run efficiently on a uniprocessor machine. The system becomes more complex as we add features to be able to reproduce the Prolog behaviour. A system that exploits parallelism explicitly can allow for more parallelism, because the user needs to program concurrently. However, there are two disadvantages: no portability and the user needs to care about control.

With respect to the applications, there are several sources of parallelism that can be exploited in a logic program. If we allow several goals in the body of a clause to proceed simultaneously and if we have enough processors, we are exploiting *and-parallelism*. If a goal matches more than one clause head and we allow several matching clauses to proceed at the same time (having enough processors), we are exploiting *or-parallelism*. And-parallelism can be further classified into *independent and-parallelism* and *dependent and-parallelism*. Parallelism exploited at this level is classified as *control-parallelism*. Another kind of parallelism that can be exploited in logic programming systems is *data parallelism*. In this scheme instead of exploiting and-parallelism and or-parallelism, we exploit parallelism that originates from the data. These are *unification* parallelism, *path* parallelism and *recursion* parallelism. Data parallelism is very fine grained, and therefore perhaps difficult to achieve good performance in parallel systems. Nevertheless, many researchers pursue the objective of exploiting data parallelism.

Unification parallelism is achieved when we allow unification of several arguments of a goal to be matched at the same time. We have path parallelism when we allow multiple unification to proceed in parallel for each alternative clause for a goal. This is different from unification parallelism in that only one argument is chosen to be unified with several clauses, although a model could be envisaged to combine the exploitation of path parallelism with unification parallelism, and even the combination of these kinds of parallelism with or-parallelism. Path parallelism is also different from or-parallelism because branches are not executed in parallel. This model also implements a search strategy different from Prolog, that is to do a breadth-first search instead of depth-first search. Continuation goals can proceed for several binding environments generated for the solutions collected in breadth-search. Recursion parallelism is very similar to parallelisation of loops in Fortran and C. Recursive clauses are unfolded and different levels of the recursion are performed in parallel.

A parallel logic programming system can also implement a form of co-routining by allowing concurrency. In fact, many Prolog related languages take advantage of concurrency to achieve better performance.

3.1.2 What kind of Architecture to Choose?

Regarding the machine architecture, one can implement a parallel logic programming system in any architecture as long as the system runs *efficiently*. This means to suit the system model to the architecture chosen. In a shared-memory machine, the implementor can take advantage of the global shared space to implement synchronisation among processors, but the fact that every processor competes for the same shared memory location is a bottleneck that can degrade the system's performance if we have many processors in the system. In order to avoid memory contention, the objective is to reduce the global shared data structures to a minimum, and also reduce task switching to avoid cache corruption.

The kind of parallelism chosen to exploit also indicates the type of architecture to use, in order to run the parallel logic programming system efficiently. For example, exploitation of or-parallelism can be efficiently implemented in a distributed memory machine as well as in shared memory machines, because or-parallelism demands very little communication among workers. Exploitation of independent and-parallelism can also be implemented in distributed memory machines, for the same reason. Tasks generated through exploitation of or-parallelism and independent and-parallelism are coarse grained and demand little communication. On the other hand, exploitation of dependent and-parallelism is more difficult to be implemented efficiently in distributed memory machines, because it requires communication between goals that share variables. In and-parallel systems that exploit determinate and-parallelism, the efficient implementation in distributed memory machines is even more difficult, because of very fine grained determinate goals.

To use shared memory architectures to implement logic programming systems has an advantage that is there is no need for replication or partitioning of the program code. In purely distributed memory machines, the program either is replicated in each memory of each processor, or is partitioned in order that each processor works on a part of the program code. A shared memory machine is also more convenient to implement models that allow exploitation of fine grained parallelism.

Load balancing and memory management become very complicated in distributed memory machines, as processors may need to access remote memories.

Dedicated architectures are used by the Japanese researchers in the context of the Fifth Generation Computer System ten-year project [80]. These dedicated machines have KL1 as their target language [121]. Their operating system is written in a concurrent logic language [19].

3.1.3 Implementation Problems and Limitations

Some of the implementation problems in logic programming are directly related to the kind of parallelism chosen to be exploited. Some of these problems are intrinsic to the model such as: maintenance of multiple bindings, variable binding conflicts, data dependency, and correctness with respect to Prolog execution.

Some other problems are related to parallel systems in general. One of them was already mentioned before: synchronisation and use of global/shared variables in shared memory machines. Load balancing is another very important problem which is common of any parallel system. In parallel logic programming systems, load balancing becomes a more complicated issue, because logic programs by nature have an irregular computational structure. By irregular we mean that logic programs are not suitable for partitioning and do not have loop structures, except in recursive predicates. Also the amount of parallelism varies dynamically with time. In the scope of recent systems that exploit both and-parallelism and or-parallelism a new problem arises that is how to distribute and-work and or-work among the processors.

Yet another issue, present also in parallel systems other than logic programming systems, is the utilisation of the system by many users, i.e., different applications that compete to use the available processors.

3.2 Parallel Logic Programming Models

Next we discuss two forms of parallelism existing in logic programming, control-parallelism and data-parallelism, and concentrate on models that exploit control parallelism. We discuss models and systems that exploit and-parallelism only, or-parallelism only, and a combination of both. In order to show the difference among all computational models, we give an example of a Prolog program that presents all forms of parallelism mentioned. Program 3.1 defines a database and is an adaptation from [27]. Or-parallelism arises in this program from simultaneous execution of clauses for the predicate `paper/4`. And-parallelism arises from the simultaneous execution of goals in the body of the clauses for `paper/4`. Path parallelism arises from the evaluation of `loc/3` for all solutions of `date/2`. Unification parallelism arises when all arguments of each goal call are matched at the same time. Recursion parallelism is achieved when the levels of the recursion `publishers/2` are executed in parallel.

```
paper(P,D,I,T) :- date(P,D), author(P,A), loc(A,I,D), title(P,T).
paper(P,D,I,T) :- tr(P,I), date(P,D).
paper(xform,1990,bristol,T) :- tr(xform,T).

author(ppro,khayri).              date(ppro,1990).
author(pprom,warren).             date(pprom,1990).
author(ppro,yang).                date(ppro,1991).
author(orsch,beaumont).           date(orsch,1991).
author(ai,costa).                 date(ai,1991).

title(ppro,muse_prolog_system_and_performance).
title(pprom,aurora_parallel_prolog_model).
title(fp,can_programming_be_liberated).
title(ppro,implementation_basic_andorra_model).

loc(khayri,sics,1990).            journal(fp,cacm).
loc(yang,bristol,1991).
loc(beaumont,bristol,1991).       tr(db,edinburgh).
loc(costa,bristol,1991).

publishers([X|Xs],[P|Ps]) :- author(I,X), publisher(P,I), publishers(Xs,Ps).
publishers([],[]).
```

Program 3.1: DATABASE PROGRAM (1).

3.2.1 Control Parallelism

Or-Parallelism Or-parallelism is exploited when we allow multiple clauses of the same predicate to be tried in parallel. For example, given the query `?- paper(P,1990,I,T).`, if we allow or-parallel execution of predicate `paper/4`, the execution of program 3.1 will generate an *or-tree* as the one shown in figure 3.1.

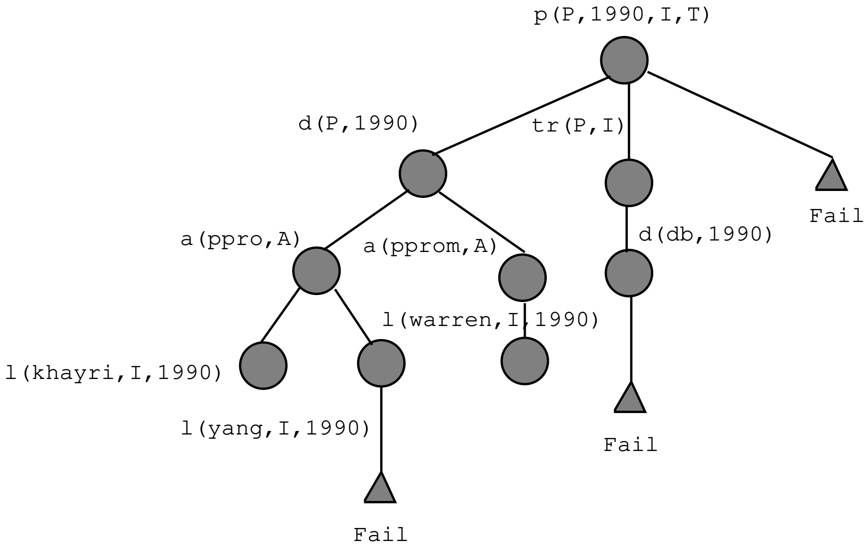


Figure 3.1: OR-TREE FOR DATABASE EXAMPLE

The tree has three branches, one for each clause of `paper/4`, that can be solved in parallel provided we have enough processors. Note that variable `P` can be instantiated to three different values for each branch. Therefore the fundamental problem to be solved in an or-parallel logic programming system is the maintenance of multiple bindings for the same variable in different branches of the search tree. As a goal can match to several different clauses and produce different bindings for a variable, a system that allows several clauses to proceed in parallel needs a mechanism to avoid conflicting bindings. In sequential Prolog this problem does not arise, because each branch is executed at once and the conditional bindings are annotated on the trail stack to be unbound later on backtracking. In parallel, each processor needs to have its own copy of the conditional variable, as processing of each branch can be done separately and simultaneously with other processors.

The main techniques used to solve the problem of conflicting bindings for variables in or-parallel systems are: *hash windows* [11], *variable importation* [86], *directory trees* [20], and *binding arrays* [138], as discussed by Crammond [30]. Each technique has an associated non-constant-time cost that is incurred at task creation time, at variable access time, or at task switching time. All three operations cannot be performed in constant-time [60]. Therefore in order to reduce the overhead costs, systems need to reduce these three costs to as low as possible. It is possible to maintain variable-access and task creation at a constant time cost. Unless the system employs a static scheduling strategy to execute the programs, task switching is always non-constant time, and in this case, it becomes one of the critical parts of the system to be optimised. From the techniques we study here, binding arrays is the only one that keeps both variable access and task creation at a constant time cost.

The idea of hash windows is based on the observation that unification of a clause head with its goal arguments, generally, only binds a small number of variables in the ancestor environments. Thus, these variables' bindings can be stored in a small "window" which is only accessible to the process that created the window and its descendant or-processes. Windows are linked for each process and are visible between choicepoints (see figure 3.2). The disadvantage of this method is that the access time for each variable is non constant, because when a descendant process does not find the variable on its own window, it needs to scan its ancestors' windows until finding the variable.

Variable importation (see figure 3.3) is based on the idea that the only variables in the ancestor environment which can be affected by unification are those which are unbound. If these variables are "imported" into the local frame for a particular process, the unification can only update variables in the local environment, independently of other or-parallel processes. The difference between variable importation and hash windows is that in the variable importation method, variables are searched for in environments local to the processor, rather than in special variable tables.

Directory trees (figure 3.4) are based on the idea that only those environments that contain unbound variables need to be copied to create a separate environment. Ancestor environments which contain no unbound variables can be shared between processes, while copies are made of other environments, with the relevant entries in the directory being changed to point to the new copy.

The binding arrays technique (figure 3.5) creates for each processor a contiguous array of entries for *conditional* variables, where conditional variables are all variables that were unbound after the creation of a choicepoint. Whenever a process needs to trail a variable, it caches its value into the binding array. Every process has the same entry to the same variable. Task switching becomes non-constant with this scheme, because in order to move on the tree, processors need to update their BAs.

The four techniques discussed to solve the problem of conflicting bindings for variables assume that workers share the main data structures of the system. There is another way

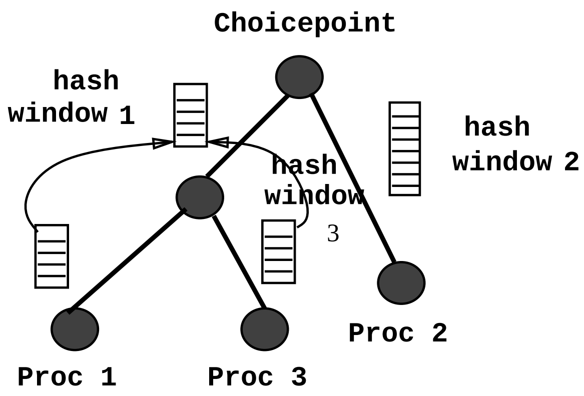


Figure 3.2: HASH WINDOWS TECHNIQUE

of maintaining multiple environments for variables that is by doing stack-copying. In the stack-shared technique processors share all stacks, but maintain independent conditional bindings on local data areas, hash-windows, directory trees, imported frames or binding arrays. In the stack-copying technique, processors do not maintain any of these structures. Instead they copy all stacks till the node they take work from. This way, they keep their own independent environments sets to produce multiple bindings to a variable. The advantage of using stack-sharing is that there is no duplication of amounts of memory used. This can be a crucial factor when combining or-parallelism with and-parallelism. The disadvantage is that the processors need to navigate through the local and global structures in order to move from one part to another of the tree. The advantage of using a technique as stack-copying is that there is no overhead on looking up of variable bindings, because they are local to the processors. The disadvantage is that combining or-parallelism using stack-copying with and-parallelism can require a huge amount of memory.

Or-parallel systems can support full Prolog, in which case they need to preserve the Prolog semantics, or can implement other language, in which case the user needs to be aware of the resulting new semantics for the execution. If the or-parallel system chooses to support full Prolog, processors need to synchronise in the presence of extra-logical, meta-logical and side-effect predicates, otherwise, the order of execution will not be the intended order. For example, if we have two clauses as in program 3.2, in order to obtain the Prolog order of solutions, it is mandatory that the processor responsible for clause 1 publishes the solution before the processor responsible for clause 2. If the query is `p(X,Y)`, and the first clause reads `2`, the answer is to write `3` on the screen. This will be the first solution. If in the second solution the clause reads `3` again, the solution will be to write `5` on the screen. This

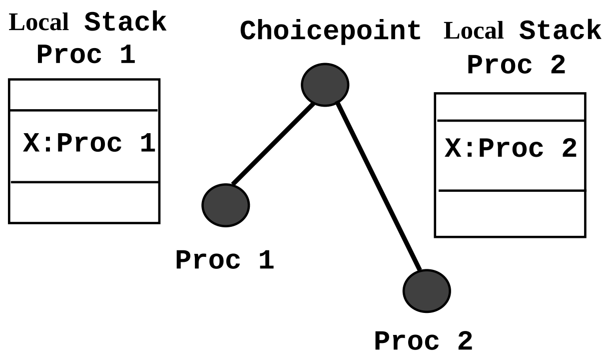


Figure 3.3: VARIABLE IMPORTATION TECHNIQUE

way the order will be 3, 5. If the second clause is allowed to produce the solution before the first one, this might affect the operational semantic given by the programmer.

```
p(X,Y) :- read(X), Y is X + 1, write(Y).
p(X,Y) :- read(X), Y is X + 2, write(Y).
```

Program 3.2: PROLOG PROGRAM WITH SIDE-EFFECTS.

If the or-parallel system gives a choice to the user to use asynchronous side-effect predicates, i.e., predicates that can be executed immediately without obeying the Prolog semantics, then the system needs to still guarantee that those side-effect predicates are going to be executed without being endangered by a cut. In other words, a side-effect predicate can be allowed immediate execution, if the branch that contains that predicate is not going to be cut later.

Examples of systems that exploit or-parallelism include the Kabu-Wake system [45], Opera [14], BRAVE [99], ROPM [78], Aurora [88], Muse [1], and Delphi [23]. Aurora and Muse are two very mature implementations of or-parallel Prolog systems. Aurora's design is based on the SRI model [138]. It uses the binding arrays technique to solve the variable bindings conflict. All the stacks are shared, but processors keep a copy of bindings for conditional variables in a contiguous block of memory.

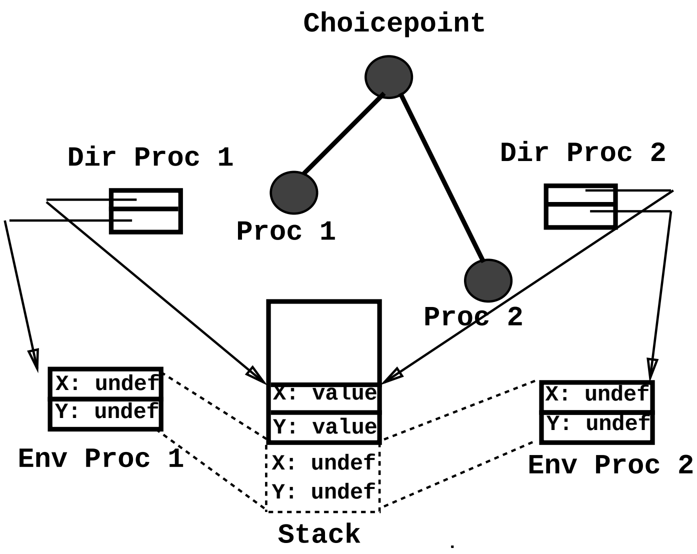


Figure 3.4: DIRECTORY TREES TECHNIQUE

Muse uses a different approach to implement or-parallelism by using stack copying whenever a processor starts working on a new alternative taken from a neighbour processor. The problem of conflicting bindings is solved because each processor works on its own stacks. The time overhead of using binding arrays in Aurora is around 25% while Muse does not incur any overhead of maintaining multiple bindings for the same variable. Both Muse and Aurora create a fixed number of processes in the beginning of the execution, which is an advantage for each process works as a pure sequential engine and keeps all the efficiency offered by the sequential implementation techniques. Ideally, one process is allocated to one processor in order to avoid heavy context switching in each processor. Both systems also handle side-effect predicates by obeying the Prolog semantics. In Aurora the default is to obey the Prolog semantics, but the user can define an 'asynch' predicate that is executed as soon as it is found, unless it is in the scope of a cut.

The Delphi model exploits or-parallelism by allowing several processing elements in a network to compute one branch of the tree independently. The idea is that if there is any machine idle in the network, this will take the program from the beginning (recomputing parts of the search tree) until it gets to a branch not yet taken by any other machine. The computation is independent for each machine in the network.

Brave is another or-parallel system developed in the university of Essex by Reynolds and Kefalas [99]. In this system or-parallelism is exploited with the aid of annotations

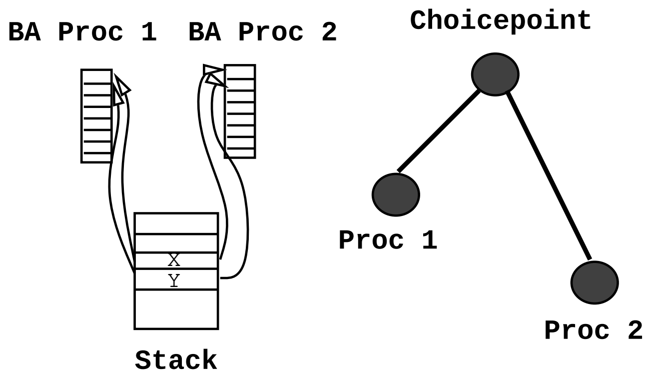


Figure 3.5: BINDING ARRAYS TECHNIQUE

provided to the programmer at the language level. The BRAVE language assumes that the programmer has control over the parallelism to be exploited. In order to deal with multiple binding environments it employs a scheme based on D. S. Warren [134] that is very similar to the SRI model suggested by D. H. D. Warren [138]. BRAVE spawns more processes than there are processors available, because they chose a model to run on a distributed memory machine where it is beneficial to have a processor busy doing context switching while waiting for communicating with another processor.

The Kabu-Wake model was designed to run in a distributed memory machine. The Opera computational model is mainly inspired from the Kabu-Wake model, first system to copy stacks when a task is created and to date the variable bindings to discard invalid bindings. Scheduling is distributed and performed by otherwise idle processors. Processors having work to share are interrupted and copy their state to idle ones. The implementation, on a specialised distributed memory architecture is based on a rather slow interpreter and provides linear speedups for large search space programs.

And-Parallelism In and-parallel systems the main problems to solve are to decide what variables are independent or dependent, which goals can be runnable immediately and how to deal with suspensions. Independent and-parallelism is exploited when the parallel system guarantees that two or more goals in the body of the clause do not share any variables. If the arguments of the goals are found to be independent at runtime then the goals are allowed to proceed independently. A system that implements and-parallelism generally has the following steps:

- an ordering phase that is needed to detect dependencies among goals;
- a forward execution phase, whose objective is to select goals for and-parallel execution; and
- a backward execution phase that is activated whenever a goal fails.

The ordering phase can be done in several ways. It can use user defined *modes* to indicate dependencies among goals. Arguments of clauses are annotated as being "input" or "output". Another method is to do static data dependency analysis of goals in the clauses. This is done at compile-time and annotates the variables as being independent. This method may lose much parallelism, with the advantage of having minimum runtime overhead. Another method is to build, at runtime, dependency graphs. This method incurs a very high runtime overhead, but can exploit all parallelism available. The final method is to generate code for dependency checking at runtime. This combines static data dependency analysis with runtime dependency checking. It is a compromise solution. Compile-time tools have been built to implement it [96].

The execution model of a system that implements IAP can be done by allowing all independent goals to produce all solutions and then *combine the solutions*, or *recomputing* all goals to the right for every solution produced by goals to the left.

One example of a system that exploits independent and-parallelism is Hermenegildo's &-Prolog [68]. In &-Prolog, processing elements called WAM agents cooperate with each other to execute independent goals in parallel. The Prolog programs are annotated with *extended conditional graph expressions* (ECGE, based on DeGroot's CGE [36, 37]) that are executed at runtime to check for groundness and independence of variables. The ECGE graphs are tests annotated to the Prolog source code at compile-time and checked at runtime. The generation of the ECGEs can be a problem, because many CGEs are possible for the same clause. The number of variables to be checked for independence can also be a problem. Groundness checking can introduce a significant overhead to the system. &-Prolog uses global analysis to infer as much information as possible from the programs [96], and produce reasonable annotated code. The system implements goal recomputation instead of combination of solutions. The example shown in program 3.1 will generate an And-tree of the kind shown in figure 3.6, with $(date(P, D)\&loc(A, I, D)), (author(P, A)\&title(P, T))$ being a possible CGE for the first clause, where & means that goals `date/2` and `loc/3` and `author/2` and `title/2` can be executed in parallel, because they are completely independent, they do not share any variable. If cross-product is used, there is no need for recomputation and goal `loc/3` can be executed for each solution tuple. This increases the quantity of parallelism exploited, but the cross-product algorithm can be a bottleneck. Cross-product can also be the source of an incorrect Prolog execution order, since solutions are produced in any order. If recomputation is used only the first solution is found in parallel. All the other solutions will be found by backtracking. Parallelism in this case is limited by the number of solutions each goal has.

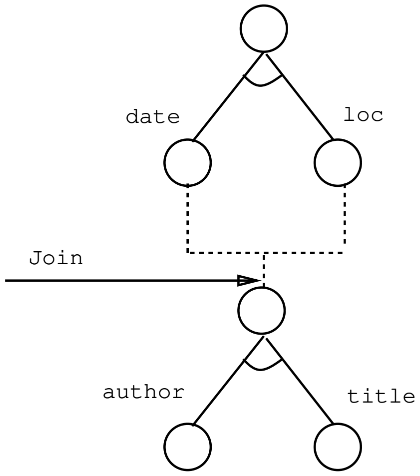


Figure 3.6: AND-TREE

Another system that also exploits IAP is APEX (And-Parallel EXecution of logic programs) [83, 84]. This model exploits and-parallelism on shared-memory architectures. The major features of the implementation are (1) dependency analysis between literals of a clause is done dynamically without incurring excessive runtime overhead; (2) backtracking is done intelligently at the clause level without incurring any extra cost for the determination of the backtrack literal; (3) the implementation is based upon the WAM, hence retains most of the efficiency of the WAM for sequential segments of logic programs. Performance results on a Sequent Balance 21000 show that APEX can achieve linear speedups on dozens of processors.

Dependent and-parallelism allows goals that share variables to go ahead simultaneously, but under some conditions. Usually, it is exploited in concurrent logic languages. The Parlog system [54] exploits dependent and-parallelism in a producer-consumer fashion. If two goals share a variable, one is considered to be the producer for the variable and the other one will be the consumer. The goal is selected to be a consumer or a producer through *mode declarations*, i.e., the programmer annotates the program with respect to the mode (input or output) of the arguments for each predicate. As long as an output argument is assigned one value, the other goal (the consumer) can start execution. If the variable is instantiated to a list or structure, the consumer goal can start to consume the partially instantiated list or structure. A subset of languages that exploit DAP do not support backtracking, such as Parlog. These are called committed-choice languages. Only one clause is allowed to be chosen through the evaluation of *guards*, usually implemented as calls to built-in predicates that make clauses mutually exclusive. Once a guard succeeds, the system commits to the clause containing the guard, and all other clauses are ignored. Other examples of these systems include GHC [126], FGHC [127], Concurrent Prolog [128, 74] and KL1 [123].

Concurrent Prolog is a logic programming language in that a program is a collection of universally quantified Horn-clause axioms, and a computation is an attempt to prove a goal – an existentially quantified conjunctive statement – from the axioms of the program. The goal statement describes an input/output relation for which the input is known; a successful (constructive) proof provides a corresponding output. The difference between Concurrent Prolog and other logic programming languages (e.g., pure Prolog) is in the mechanism they provide for controlling the construction of a proof. Prolog uses the order of clauses in the program and the order of goals in a clause to guide a sequential search for a proof, and uses the cut operator to prune undesired portions of the search space. Concurrent Prolog searches a proof in parallel. To control the search, CP embodies two familiar concepts: guarded-command indeterminacy, and dataflow synchronisation. They are implemented using two constructs: the commit operator and the read-only annotation.

KL1 is a system based on the KL1 language, that in turn is based on GHC (Guarded Horn Clauses). In these languages the control is also added by the user through *guards* that guarantees only one clause to be committed. The KL1 language was designed in the context of the Fifth Generation Computer Systems Japanese project and is being used as the kernel language of the PIM machines family [105].

Yang [144] designed a language, P-Prolog, based on mutual exclusion relations where a set of clauses can be "backtrackable". Clauses in a predicate in this language are grouped. Clauses in the same group behave in the same way as GHC, but if a fail occurs for a group of clauses, another group can be tried. The P-Prolog language introduced a new concept of programming in committed-choice languages, by allowing a degree of don't know non-determinism to the programs.

Another system that exploits dependent and-parallelism, but in a different fashion is DDAS [109]. In DDAS the variable dependency is checked at compile-time and the program is annotated with extended CGEs similar to the ones used in &-Prolog. Goals to the right of a conjunction are allowed to proceed in parallel as long as they do not bind the dependent variable. Only the leftmost goal is allowed to bind the dependent variable in order to keep the operational semantics of Prolog. Hence rightmost goals that try to bind a dependent variable are suspended until they become leftmost in the CGE.

In the program of figure 3.1, dependent and-parallelism is obtained by building the graph shown in figure 3.7. All goals are launched in parallel. The goals `date/2`, `author/2` and `title/2` try to bind the variable `P`, but only `date/2` is allowed to produce a binding for the variable `P`. Therefore, `author/2` and `title/2` suspend while waiting for `date/2` to produce a binding for `P`. As soon as `date/2` produces a value for `P`, it wakes up goal `author/2`, and goal `title/2`. The same happens to the goal `loc/3` that tries to bind variable `A`. It needs to suspend waiting for `author/2` to produce a value for `A`. Execution follows like that until a solution is found, in which case a kind of semi-intelligent backtracking takes place in order to find other solutions.

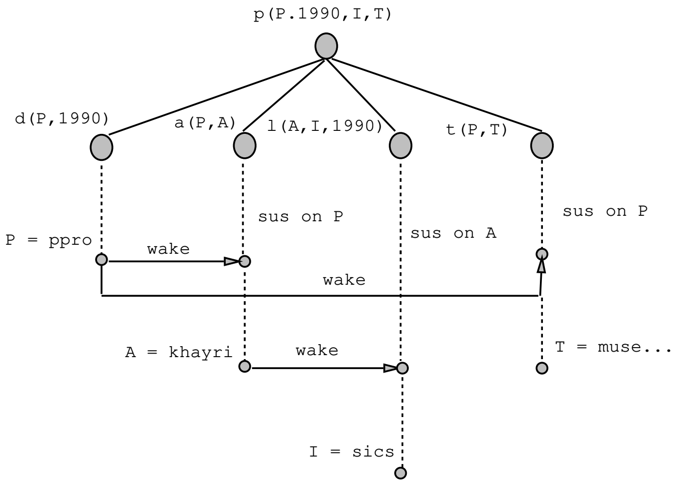


Figure 3.7: DDAS EXECUTION TREE

And- and Or- Parallelism Combined Other models and systems are designed to exploit both and- and or-parallelism. The first model known to incorporate and- and or-parallelism in a single framework is Pollard's [98]. In this model, or-parallelism is combined with independent and-parallelism, and solutions are found by combination and not by backtracking and recomputation. Following Pollard's work, Conery designed a model based on processes and message passing that aimed at exploiting also independent and-parallelism and or-parallelism. It is known as the And-Or Process Model [26]. In Conery's model, dependent and independent and-parallelism are exploited together with or-parallelism. The main idea is to create processes for each and-goal and each or-alternative. These processes communicate with each other via message passing. Messages of *start*, *redo* and *cancel* are sent downwards (i.e., from a parent process to their children). Messages of *success* and *fail* are sent upwards. A *start* message is sent to the first process in the beginning of the computation. The *redo* messages are sent from an and-process to an or-process to ask for another solution (in this model the or-process works as an oracle producing solutions in advance). A *cancel* message is sent whenever a process reaches a state where it will never use any *success* messages. A *success* message is sent whenever an or-process finds a solution. A *fail* message is sent when a process cannot solve the problem. In this model the order of execution of goals is maintained by a literal ordering algorithm that generates a dependency-graph at compile-time and updates it at runtime. This model will generate the following literal ordering for the program 3.1:

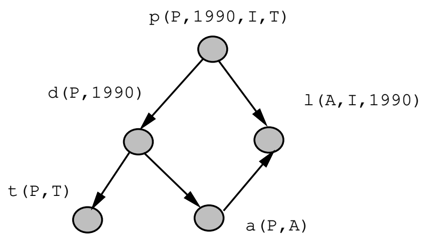


Figure 3.8: LITERAL ORDERING FOR CONERY'S EXECUTION MODEL

PEPSys [142] is another example of system that exploits both and- and or-parallelism. And-parallelism is exploited in a restricted way, where only goals that are **strictly** independent can proceed in parallel, and just and-parallel deterministic computations are allowed in parallel. PEPSys implements independent and-parallelism by means of program annotations that indicate which goals are to be executed in parallel. It implements or-parallelism by using the hash-windows technique. When combining and- and or-parallelism, processes are created to execute the cross-product solutions. Not much is reported about combination of and-parallelism with or-parallelism. PEPSys deals with cuts by introducing a new built-in predicate `one-solution/2`. This assures that only one solution will be exhibited.

LORAP [10] is another model that combines limited or-parallelism with restricted and-parallelism, and was designed for distributed memory machines. Restricted and-parallelism is obtained when we impose restrictions to execute goals in parallel. For example, CGEs restrict the parallel execution of some goals. The system is very similar to Conery's and/or process model, although there are significant differences in the way LORAP achieves its goals. For example, in order to control explosion of processes, LORAP implements a redo message on demand. Also there is no notion of gathering solutions. Process-based systems as LORAP or the and/or process model do not present any overhead of task switching given that a process is created for each new goal to be executed. On the other hand, they pay off on process creation and maintenance.

The Reduce-Or Process Model [78] is also similar to Conery's model, but also can exploit dependent and-parallelism through the Data Join Graphs (DJP) that allows to represent dependencies between goals, contrarily to the literal ordering graph used by Conery's model, and DeGroot's CGEs.

The Basic Andorra Model [139, 65] combines dependent and-parallelism in the style of committed concurrent systems, with only determinate goals being allowed to be executed in parallel, with or-parallelism in the style of Aurora. An implementation of the Basic Andorra Model, Andorra-I [102, 148] is the main subject of discussion in this thesis.

The Extended Andorra Model [140] whose design comprehends the exploitation of all kinds of parallelism (or-, IAP and DAP, determinate or non-determinate) in a single framework is another system that aims at combining and- and or-parallelism. Gupta [64] showed that an implementation of the Extended Andorra Model is feasible by implementing an interpreter to a subset of the model.

Prometheus [108] is another model that exploits or-parallelism with dependent and-parallelism in the style of DDAS.

ACE [59] is a model that exploits or-parallelism as in Muse with independent and-parallelism as in &-Prolog. A version of the system that supports independent and-parallelism is already implemented [61].

IDIOM [62] combines or-parallelism with deterministic dependent and-parallelism and independent and-parallelism. There are at least two approaches to implement independent and-parallelism in the IDIOM model. The first one consists of *recomputing* solutions whenever we try to find the next solution for a goal. The second approach is to allow all independent goals to proceed in parallel and *combine* the solutions, using a time-stamp based algorithm [56, 57]. The advantage of using recomputation is that it imitates Prolog execution and maintain compatibility, while keeping lower overheads of implementation. The disadvantage is waste of memory, because each recomputation requires repeated amounts of memory, and loss of parallelism. Combining solutions has the advantage of allowing for more parallelism, but the algorithm to combine solutions can be very complex because of the number of variables to be cross-checked, and number of solutions for each goal.

Most recent parallel logic programming systems that combine and-parallelism with or-parallelism are ParAKL [92] and Penny [91]. Both systems are based on the Extended Andorra Model, without implicit control. The systems assume that the programmer has control over the parallelism by using language constructs to guide the exploitation of parallelism.

ParAKL has its main execution mechanism based on limiting the amount of non-determinism in the Andorra model. This is a very important issue in logic programming systems, as creation and maintenance of choicepoints is very expensive. The system uses intelligent selection criteria, instead of doing leftmost selection of a goal to create a choicepoint. The criteria to select the goals to do non-determinate promotion are (1) select the goal that have fewest alternatives, and (2) select the most constrained goal.

Penny is an implementation of the AKL language whose design keeps the selection of leftmost goals to make a choicepoint. Non-determinate goals that share variables will execute until one of the goals tries to bind the shared variable. At this point on only that goal is allowed to proceed and the others are suspended until the initial goal produces a solution. As in Andorra-I, the language also incorporates constraint programming features, which adds another degree of complexity to the combination of and-parallelism and or-parallelism

Several implementation methods are used to combine or- with and-parallelism. The ACE model, for example, mixes techniques previously used by the Muse system and &-Prolog to exploit both or- and and-parallelism. The Paged Binding Array [63] is another technique to combine and- with or-parallelism, whose main idea is based on the SRI model's binding arrays [138] and resembles the pagination operation in an operating system. It mixes the techniques already successfully used in the Aurora and &-Prolog systems.

In the next section we give a brief description of some parallel logic programming systems that exploit data-parallelism.

3.2.2 Data-Parallelism

Or-parallelism of a highly regular nature can be exploited as SIMD parallelism. There is only one control thread, but multiple environments. This kind of exploitation of parallelism is useful when there are variables that take different values. Multilog [114] uses *variable vectors* to implement multiple bindings, while Multipath [52] uses binding environments. Reform Prolog uses unfolding of clauses to exploit parallelism at recursion level, and Succi's SEL exploits unification parallelism.

Unification parallelism can be exploited implicitly and is implemented by allowing several arguments of the head of a clause to be unified in parallel. This leads to very fine grained parallelism that requires special machinery to perform well. An example of system that exploits unification parallelism is Succi's SEL system [116].

One example of a system that exploits data parallelism at recursion level is the Reform Prolog [9]. Reform Prolog is a model that relies very much on global analysis techniques to help unfolding recursive calls and transform them in loops that can then be parallelised as a FORTRAN or C loop, originating data-parallelism.

We concentrate on this thesis in describing models and implementations of parallel logic programming systems that exploit and- and or-parallelism, either separated or combined, therefore we will not go into detail about systems that exploit unification parallelism.

3.3 Discussion

As was mentioned before, parallelism can be exploited transparently or can be exploited with the help of the programmer. Some extensions of Prolog exist that include control primitives for parallelism. For instance, Parlog and AKL are implementations of logic programming languages where the control is very much left to the programmer, while other systems aim at exploiting parallelism completely transparently, as the Basic Andorra Model, DDAS or &-Prolog. In those systems where some control is added to the language in order to extract parallelism, we need a new programming language, which makes programs written previously in Prolog not portable. On the other hand, new languages may be more suitable for concurrent programming because the programmer can state where the parallelism is. We argue that this design approach may be (with restrictions) suitable for systems implementor, but it is definitely not the best approach to be taken if the parallel system is going to be used by the average user that only wants to use the language as a nice developing tool for solving problems.

Parallel logic programming systems can produce a single solution or many solutions (except committed choice based systems that are deterministic and produce only one solution). When trying to find only one solution the parallel systems need to concentrate resources on the path that will most likely lead to the solution. When trying to produce many solutions the parallel system needs to spread resources among all the work available on the tree.

Because of the nature of logic programs where the parallelism varies irregularly during the execution, scheduling plays an important rôle on the efficient execution of parallel logic programming systems. It will be the decision of the scheduler to concentrate resources on one part or another of the execution tree. More than that, scheduling needs to guarantee the correct execution of Prolog programs containing pruning operators or side-effect predicates. In some systems, such as &-Prolog, a good scheduling strategy is also needed in order to allow correct backward execution of body goals, and avoid the problem of trapped goals [111]. A scheduler also tries to use resources efficiently, i.e., being fair with other users in the system and not taking more resources than needed.

In this chapter, we discussed forms of exploiting parallelism in logic programs. We concentrated on control-parallelism and parallelism exploited implicitly. We described several techniques to implement and-parallelism and or-parallelism, and techniques to combine and-parallelism with or-parallelism. It is clear from all the models and systems that there is an issue not yet discussed and that is very important: *work distribution*. We saw that several of the techniques described make task switching a non-constant time operation. Therefore, the issue now is to reduce the time and frequency of task switching, and therefore to increase the overall performance of the parallel system. Moreover, the new recent models that aim at exploiting both and- and or-parallelism presents a new feature that is to produce two kinds of parallel work at the same time. This leads to a new problem, which kind of work to choose at runtime? At the end of this work we will give a solution to this problem.

3.4 Summary

In this chapter we discussed the main issues involved in adding parallelism to logic programming systems. We discussed issues such as the kind of parallel machine architecture to be used, problems of implementation, and limitations to exploit parallelism in logic programs. We presented several parallel logic programming systems and discussed their implementation issues. We concentrated on how to exploit *or-parallelism* and how to exploit *dependent and-parallelism*, and *independent and-parallelism*.

We described models and systems that exploit and-parallelism only (dependent or independent), or-parallelism only and discussed the combination of both kinds of parallelism. When describing the implementation models, we discussed the most used techniques to support or-parallelism such as binding arrays, hash windows, directory trees and variable importation, most used techniques to implement independent and-parallelism such as recomputation, combination of solutions and backtracking mechanisms, and most used techniques to implement dependent and-parallelism such as suspension handling, and backtracking mechanisms. In this chapter we also discussed issues in combining and-parallelism (dependent and independent) with or-parallelism, and briefly described the target system used in this thesis, *Andorra-I*, that combines *determinate* and-parallelism with or-parallelism according to the *Basic Andorra Model*.

Lastly, we discussed an important component that every parallel system must have: the *scheduler*. The lack of schedulers to distribute and- and or-work in systems that exploit both and- and or-parallelism is the main motivation to this research work.

Chapter 4

Scheduling in Parallel Logic Programming Systems

In any parallel system, the scheduler is a very important part because it is responsible for the efficient utilisation of processors by the system. By efficient utilisation we mean to use only the resources required by the application. A reasonable scheduling strategy will also complete a parallel application in a time close to the earliest possible time. In parallel logic programming systems, the execution of logic programs generally produces irregular computations that can be mapped into trees. Some applications produces very "bushy" trees, while others produce "thinner" trees. The trees have nodes that are distributed throughout the computing resources to be executed.

In or-parallel systems the nodes of the execution tree are or-nodes (also called choicepoints) that contain alternatives left to be executed. A collection of or-nodes connected in some hierarchical order form an *or-tree*. Finding or-work, i.e., finding new alternatives to execute, is the task performed by an or-scheduler. The *or-scheduler* deals with problems such as which alternative to select next to continue working or which branches need to be pruned.

And-parallel systems generate the same kind of trees as or-parallel systems, but with and-nodes. A collection of and-nodes connected in some hierarchical order form an *and-tree*. In and-parallel systems, the *and-scheduler* is responsible to select the next goal to be executed.

In systems that exploit both and-parallelism and or-parallelism there is a new issue to be tackled which is *what kind* of work to be chosen next: and- or or-? There is not much research on this scheduling issue, because the combination of and-parallelism and or-parallelism is a relatively new issue and there are not many implemented systems that exploit both and-parallelism and or-parallelism. Yet another issue to be tackled, and this is applicable also to and-parallel and or-parallel systems, is the *efficient* utilisation of resources, i.e., utilise only the resources required by the application. This is a very important issue in multi-user parallel environments.

Systems that exploit both and- and or-parallelism usually generate *and-or trees*, although several variations of exploitation of and-parallelism may lead to different conceptual trees.

A *task* to be selected by a scheduler to allocate to a processor is any work done by the system engine between two invocations of the scheduler. In that case, depending on the implementation system, the task sizes may vary. A *task switch* is performed when the scheduler is invoked to find work, and the worker actually switches to another task.

Next we discuss some or-only and and-only scheduling techniques used for some well known parallel logic programming systems that exploit either and-parallelism or or-parallelism, and and/or scheduling techniques proposed for systems that exploit both and-parallelism and or-parallelism.

4.1 Scheduling Or-Parallel Work

In a parallel logic programming system an or-scheduler has the function of choosing the best choicepoint from which to take an alternative to be explored, in order to utilise resources in a better way. For both all-solutions and single-solution applications the scheduler should, in an abstract level, keep the height of the execution or-tree to the minimum possible (in order to achieve speedups related to the sequential execution). From a practical and concrete level, it should not incur too high costs in task switching, and it should cope with a limited number of processors in the system. Another very important issue is to reduce the amount of interaction between the workers. Some schedulers also deal with the problem of "speculativeness". This means that whereas some branches of the search tree are to be executed compulsorily, others (speculative or-branches) might be pruned because of a cut or commit operator. A branch of the or-tree can also be speculative if we can find ways of knowing that it does not lead to a valid solution.

Several schedulers have been proposed to distribute or-work in or-parallel logic programming systems. In shared memory systems implementations such as Aurora [88] and Muse [1], or-schedulers try to minimize frequency of task switching, and amount of interaction between the workers. In the implementation of other models [137, 40], methods are used that keep the cost of task switching constant by maintaining the cost of task creation or variable access as non constant. Gupta [60] gives a comprehensive taxonomy of or-parallel execution models for logic programs and discusses the issues of variable access time, task switching costs and task creation time. Task switching costs are directly related to scheduling. It is known from [21] that it is more efficient to keep the cost of task switching non constant while keeping other costs – variable access and task creation – constant. This section concentrates on describing or-schedulers that work for parallel systems whose cost of scheduling is non constant. So, reducing costs of task switching is one of the main subjects we talk about in this section.

Contrary to distributed systems, schedulers for shared memory systems do not need to tackle the processor communication problem. Instead, the greatest problem to be tackled by an or-parallel scheduler is to control the frequency of task switching in systems that do not have constant time for this operation. Indirectly, two problems of equal importance arise when we do not control the frequency of task switching: cache corruption and scheduling overheads. These problems can also occur in distributed memory systems where the penalty for not maintaining locality of references is more serious, incurring memory accesses that can be 10,000 times slower than a shared memory access.

There are several different schedulers for or-parallel systems, but the main basic strategy to follow, in general, is: *to give preference to mandatory work. If there is no mandatory work select the least speculative work and repeatedly seek better work*. This is the algorithm that seems to fit all or-parallel systems nowadays and proved to be the best so far. Around this main algorithm, several different strategies implement some or all of this.

There are in the literature several strategies describing different schedulers for different models, and even for the same model. For example, the Aurora system has at least five scheduler strategies that compete with each other for parallel efficiency. The strategies vary mainly in the way that the work is chosen in the execution search tree. The schedulers also use different data structures that affect in different ways the performance of the or-parallel system. Some of them, the most recent ones, deal with speculative work. All these strategies are supposed to be dynamic, i.e., decisions are made based on information collected at runtime. There is at least one known or-scheduler design that is aided by information generated at compile-time [48]. This was implemented as a simulator and results were compared with the Aurora or-parallel system.

Dynamic or-schedulers can be classified as demand-driven or work-driven. Demand-driven schedulers are usually more efficient because they do not require that a busy worker interrupt its own work to publish newly created work. In work-driven systems workers publish their newly created work and this can affect the worker's own performance as it has to spend some time publishing its work. In demand-driven schedulers it is the responsibility of an **idle** processor to look for work while idle. It just interrupts busy processors once to take the work.

Scheduling strategies can also be classified as preemptive or non-preemptive. Non-preemptive strategies will work on a task until its completion or failure. Preemptive strategies will voluntarily suspend any task at any time. Preemption is a very important feature when dealing with speculative work, because processors can try to move to a less speculative task.

There have been many attempts to speed up the parallel execution of or-parallel logic programming systems through various or-scheduler strategies. If the parallel system assumes the same operational semantics of the Prolog language, the scheduler needs to guarantee that the parallel execution of any Prolog program will produce the same output as its sequential counterpart. This is not a trivial task, specially if we observe that many Prolog programs are written to run efficiently on a single processor, making the parallelisation task very difficult. The main problems involved in keeping correct execution of Prolog programs in a parallel environment come from the utilisation of pruning operators (cut and commit) and from side-effect predicates (I/O, for example).

The first attempts at scheduling in parallel Prolog systems did not consider the problems caused by the pruning operators and side-effect predicates. The first or-schedulers tried to address other issues such as the kind of strategy that should be used by the scheduler, bottom-most, i.e., take alternatives from the deepest created choicepoint (this implements a depth-first search); top-most, i.e., take alternatives from the nodes closer to the root of the or-tree (this implements a breadth-first search); or a combination of those. Some or-parallel engines (e.g., Aurora and Muse) divide the tree into two parts: private and public. This way each processor can happily work on its private nodes until the search is exhausted and there are no more live nodes in the private region. In this case the processor will try a public backtrack. If there is no more public work left, new or-work is made public either on demand of idle workers or according to the choice of a busy worker.

Scheduling on top-most nodes was believed to reduce the public part of the tree and therefore reducing cost of moving around. It was also believed that nodes closer to the root hold more work. The disadvantage of this approach is that it may cause too frequent task switching if the task size is small. On the other hand, dispatching on bottom-most opens large public sections, so that work can be found by backtracking (*minor task switching*) rather than by moving to another branch (*major task switching*).

Next we describe some of the schedulers proposed and used in or-parallel Prolog systems. Four of them are used in the Aurora system, one of them is used in the Muse system, and the others are used for other or-parallel systems.

The Argonne Scheduler [15]

The Argonne scheduler was one of the first attempts to overcome the problem of efficiently executing a Prolog program in or-parallel. It also dealt with data locality by distributing to each process the job of finding its own task to execute next. There is a reduced utilisation of global structures. This is a very important approach that is followed by almost all its successor or-schedulers. The scheduler was inspired by ideas of Warren [138], but did not try to address the problems generated by the execution of cuts or side-effect predicates.

The main characteristics of this scheduler are:

- the scheduler is distributed and the use of global structures is very limited.
- the nodes of the tree are divided into "public" and "private". A process exploring a node at the private part of the tree can run at the same speed as a normal sequential Prolog engine;
- the scheduler's job is restricted to coordinate the movement of workers in the public part of the tree, leaving the private part to be managed by the main engine;
- its strategy is to dispatch on topmost nodes in order to reduce costs of moving around the execution tree, and minimize the size of public regions in the tree. It also considered that top-most dispatching would distribute (heuristically) larger pieces of or-work (high granularity work).

This approach aims at minimising synchronisation between workers and to make the system scalable by not using many global structures. It failed to achieve very good efficiency because the system had a race problem in that all processors would migrate to the same piece of or-work. When the worker arrived to get the work, it may have disappeared, because another worker would have taken it before. This strategy favours a breadth-first strategy search which is good for all solutions problems, but which is not so good for programs that contain speculative work.

The Manchester Scheduler [16]

The Manchester scheduler was developed in parallel with the Argonne scheduler and within the same project. In the same way as the Argonne scheduler, the Manchester scheduler uses dispatching on top-most nodes and divides the tree into public and private parts. It differs from the Argonne scheduler in the way it selects nodes from the public part of the tree. Its main concern is with the number of bindings needed to be installed in and removed from the binding array as Aurora uses the SRI model [138] to implement or-parallelism. It maintains a public candidate node that is made public whenever the public work available is exhausted. The scheduler matches idle workers that are closer to the public candidate node. In that case, the Manchester scheduler avoids the race problem that existed in the Argonne scheduler.

This scheduler is one step ahead of the Argonne scheduler in that it deals with pruning operators and side-effect predicates.

The Muse Scheduler [1]

The Muse scheduler was the first one to implement the idea of bottom-most dispatching. It tries to deal with several kinds of applications, including those that have speculative computation. Workers only search for leftmost work when they perform voluntary suspension. If the workers are searching for work because they are idle, the search strategy is to look for the richest worker, i.e., the one that contains most live nodes on its branch. Voluntary suspension is done at time intervals given by the user. The left-to-right ordering between workers is maintained by an array indexed per worker that contains the number of workers to its left. Every worker is suspended at the time of the suspension and a number of rightmost workers try to search for work on its left by updating the array entries. The left-to-right order of the tree is maintained on demand, and not all workers perform voluntary suspension, just the n-rightmost workers will do so. No voluntary suspension is applied when there idle workers. One of the problems of the Muse scheduler is that voluntary suspension is also done on mandatory work.

The Muse scheduler seems to be very successful when dealing with speculative applications.

The Bristol Scheduler [8]

The Bristol scheduler was designed primarily in order to investigate the Muse bottom-most dispatching strategy when applied to the Aurora system. So far, only top-most dispatching was being used in the scope of the Aurora system, because it was believed that to keep the public part of the tree to a minimum would incur less overhead. As the Muse system was obtaining very good speedup results with its bottom-most dispatching strategy, the Aurora implementors started to investigate strategies for doing bottom-most dispatching. Another motivation for the design of the Bristol scheduler was to overcome some of the disadvantages and reduce the overheads existing in the Manchester scheduler. A performance study of the Aurora system [119] using the Manchester scheduler was reported and pointed out some solutions for improving the performance of Aurora.

The Bristol scheduler followed a long path before it would deal with all the problems in scheduling or-parallel work in Aurora. It first implemented different strategies that could be adjusted manually and also voluntary suspension. This was a step to try to deal with speculative work. If workers can suspend they can give up some task to start another one less speculative.

Several strategies have been used by the Bristol or-scheduler (e.g., leftmost and richest worker selection [6]), but results reported that the one that gives best speedups is the hybrid strategy [7] where **busy** workers every so often voluntarily suspend seeking for less speculative work in the leftmost part of the tree. **Idle** workers seek for work from the richest workers. Recently the Bristol scheduler introduced a new issue to the scheduler by using the pruning operators counters and scope [67] to figure out which work is speculative and which is not [8]. The main algorithm of the Bristol scheduler follows the basic algorithm mentioned before, where workers always give preference to mandatory work, and are always seeking for better work if they are working in any speculative work.

While producing good speedups for speculative programs, this scheduler also produces good speedups for programs without any speculative work.

The Bristol scheduler also tackles a problem that was not approached before by any of its predecessor schedulers. It incorporates a scheme to control resource utilisation by releasing resources whenever they are not needed, and requesting resources when they are necessary to the application. Results of observing the machine workload showed that efficiency can be much improved by using a policy of temporarily releasing workers back to the operating system if there was not enough work available.

The Dharma Scheduler [113]

The Dharma scheduler's main design goal is the efficient implementation of a strategy for speculative work, and it borrows its main ideas from the Wavefront scheduler [13]. The main goal is therefore to solve the problem of scheduling speculative work. The Dharma scheduler considers the whole tree as being speculative and tries to concentrate workers on the deepest leftmost part of the tree. It achieves efficiency by implementing a branch level dispatching algorithm instead of a node level dispatching algorithm. The branch level dispatching algorithm uses a data structure linking the tip of each branch in the public part of the tree, called the *active chain*. This structure avoids scanning nodes of the execution tree, and consequently avoids locking operations. It maintains the left-to-right order of the branches on the tree. Each element of the active chain, a tip, that represents a branch in the execution tree, contains information about its right and leftmost sibling, tip type, and information about the branch itself.

It allows voluntary suspension of workers at a given suspension interval in order that they can seek for better work to their left. The number of workers to be suspended depends on a parameter given by the user. A new implementation is on its way to try to set the suspension interval automatically by using information about pruning operators counters as used by the recent version of the Bristol scheduler.

The most recent results presented by Sindaha [113] suggest that the strategy and implementation technique utilised have several advantages over the other current schedulers that deal with speculative applications. It also behaves quite well for non-speculative applications. The advantages are that, by using the active chain, less contention is achieved when searching for work in the leftmost order. The active chain also reduces the communication ratio between workers, because workers do not need to interrupt each other to look for work in the nodes. It still needs synchronisation though to make work public. An extra overhead is introduced because the tips need to be updated for each operation of removing or adding a new branch on the search tree or for pruning operations.

This strategy and implementation seems to be the most successful so far in dealing with speculative applications as well as non-speculative applications.

Or-scheduler Techniques for other Parallel Systems Other or-parallel scheduling techniques have been proposed, but either have not been implemented or have only been simulated. Wai-Keong [48] reports an or-scheduler strategy that uses a heuristic task distribution by assigning "weights" to the alternative clauses. Its algorithm to assign the weights is similar to Tick's granularity size algorithm [124]. In this algorithm each goal is counted as having weight 1 and each recursive call also has weight 1. These weights are assigned to the clauses at compile-time by annotating the Prolog program. The scheduler then uses this information in order to select tasks to spawn. This technique was originally used by Tick [124] to control the scheduling of and-parallel goals in FGHC, but it was used by Wai-Keong to control the spawning of or-parallel alternatives.

Another scheduling technique is reported in [85] where a method to remove structural imbalance of the programs by global analysis (basically unfolding/flattening recursive predicates) is proposed. By removing structure imbalance the author assumes that the or-work is evenly distributed during the execution. The following rules are applied to distribute work:

- *eager-splitting strategy*: at each choicepoint where m processors are present, assume there are n valid choices. m tasks are created and assigned evenly to m processors. If $n \geq m$, each task contains $\frac{n}{m}$ choices, and the remaining choices are randomly included in some of the tasks. If $n < m$, each processor randomly picks one choice.

- *lazy-splitting strategy*: at each choicepoint, two tasks are created and assigned to each of the half of the processors. In case of choices being not evenly divisible, remaining choices are treated in a way similar to that in the eager-splitting rule.

4.2 Scheduling And-Parallel Work

There are a number of systems that exploit and-parallelism in logic programs. Some of them exploit parallelism explicitly [32, 22]. Others, for philosophical reasons or in order to keep portability of Prolog programs, exploit parallelism implicitly [148, 9, 68].

Scheduling And-parallel goals in &-Prolog In systems that implement IAP the main problem in and-scheduling is to correctly implement backtracking inside a CGE. If any goal is selected to execute in and-parallel, and the implementation is WAM-based with contiguous stacks, on backtracking, a worker may not be able to backtrack and continue a previous goal, because there may be another goal being executed after the goal that needs to start again. This is shown in picture 4.1. From goals `a & b & c`, worker 1 (w1) selects goals `a` and `c`, while worker 2 (w2) selects goal `b`. Assume that worker 1 finishes goal `a` before worker 2 finishes goal `b`. In that case, worker 1 starts goal `c`. Assume that worker 2 finishes execution of goal `b`. It needs to backtrack and find another solution to `a`, but `a` cannot continue, because goal `c` did not finish yet. This is illustrated as the *trapped goal problem*.

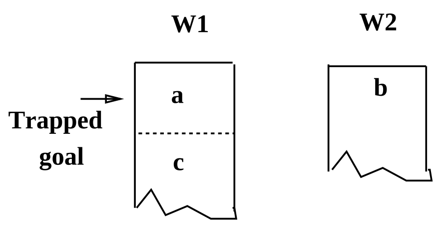


Figure 4.1: THE TRAPPED GOAL PROBLEM

The strategy currently used by the &-Prolog system is random selection of parallel goals from any agent's stack. As shown by the explanation before, this strategy is dangerous if we allow goals inside a ECGE to backtrack. In this case it can cause the problem of "trapped" goals where the topmost goal in an agent's stack backtracks on top of an older goal that did not finish execution yet. In order to solve this problem, there are three approaches. The first one [69] was designed for the &-Prolog system itself, and consists of only selecting a goal for parallel execution if it is the oldest pushed on the goal list. Therefore, WAM agents only execute goals in a certain order that is the sequential Prolog's order of goals. The second approach is to use a more flexible scheduling strategy that, instead of restricting selection of goals by age, can select any goal. This is achieved by maintaining a *separator marker* between goals that can be executed in parallel. This scheduling strategy is described by Shen [111] and used in the DDAS system. A third approach that will probably be used by the Paged Binding Arrays implementation is to employ a memory management that allocates memory segments dynamically instead of using contiguous chunks of memory.

Scheduling And-parallel goals in Parlog Parlog, as was mentioned before, is a concurrent logic language and has declarative and operational semantics different from Prolog. Users need to program using mode declarations for every clause in the program, and need to use guards to prevent don't know non-determinism.

Crammond [31] describes 3 different algorithms to schedule and-goals in the parallel Parlog system and analyses their effect on performance. The algorithms vary basically in the way the pool of goals is implemented. The three strategies are shown below:

- maintain a local run queue per worker;
- maintain local run queues with private and shared parts;
- maintain local run queues with a single global run queue.

Crammond's experiments were performed on a Sequent Symmetry machine. The paper reports that local run queues with private regions produce the best results in terms of locality and locking overhead. The amount of work taken from the local run queues is better proportioned. Also this algorithm produced the most consistent execution times.

A similar scheme was implemented in Andorra-I, but it seemed that for some programs goals that produced lots of determinate work were always being put in the private region of the local run queue, and work was not being distributed proportionally among the workers. And-scheduling in Andorra-I is discussed later in the next chapter.

Scheduling And-parallel goals in FGHC FGHC (Flat Guarded Horn Clauses) is a language based on GHC [126]. Like Parlog, FGHC is a concurrent logic language, that supports committed-choice non-determinism and stream communication. Parlog requires compile-time analysis in order to guarantee that the guards are safe, i.e., will not bind external variables.

KL1, a language based on FGHC and whose parallel implementation is the kernel language for the PIM machine, required static scheduling. Programs written in KL1 are annotated with KL1 primitives that inform the system which parts of the program code can be parallelised, i.e, the program is partitioned by the programmer. Therefore, scheduling in this system is made in a static fashion given that the user correctly partitions the program in order to be able to exploit parallelism.

Tick [124] studies several scheduling strategies to select goals in FGHC with the aid of a very simple compile-time granularity analysis algorithm. The idea of the algorithm is to assign weights to the clauses in the program. The weight of each goal call in the body of a clause is one. Also recursive predicates are assigned weight one. The scheduler of the FGHC parallel emulator running on a Sequent Symmetry machine was adapted to use this information. Demand-distribution methods are discussed, including *oldest-first* and *heaviest-first* distributions. His results did not produce good speedups due to various factors such as high system overheads, low cost of spawning a goal on a shared memory multiprocessor, and the increase in synchronisation caused by the method. His conclusion was that it does not matter what kind of selection is used to choose which goals to execute next. Newer results have not been reported in the literature.

4.3 Schedulers for Models that Combine And- and Or-Parallelism

There are few reports of systems in the literature that exploit both and-parallelism and or-parallelism [142, 78]. Most of them are not very mature systems and therefore are not yet tackling scheduling problems. Hence very few researchers have been working in the area of distributing and-work and or-work. The new issue here is *what* kind of work to distribute, and-work or or-work. So far, only the problem of scheduling for and-parallel only systems and the problem of scheduling for or-parallel only systems have been tackled. With the most recent systems that aim at exploiting both and-parallelism and or-parallelism, the new issue is what kind of work to distribute. Some solutions were found that use some kind of global analysis of the program to predict the amount of work to be done in each branch [35, 124, 82]. These algorithms work at compile time by calculating inter and intra size arguments of goals and clauses and generating recurrence equations that would be utilised by the scheduler at runtime. However these solutions have not been applied to any known parallel logic programming system.

Another solution is to consider that all Prolog applications when executed in parallel generate a certain amount of work that can be mapped onto a statistical distribution [85]. Assuming that all tasks are to be executed with the same probability distribution, an algorithm is used that obtains the probability to the branches in order to distribute tasks evenly among the processors. This technique, although applied to a model that aims to exploit and-parallelism and or-parallelism, only solves the or-parallel work distribution.

Another solution is used by Gao et al [149] and involves generating feedback information to the system by profiling previous interactive system sessions. The solution proposed tries to solve almost all problems by using a combination of static, dynamic, and hybrid approaches to get granularity sizes of Prolog programs, and feed these results into the system together with most-frequent used modes for calls to predicates. The scheduling strategy for or-parallel tasks is to choose the nodes nearest to the root. The strategy to choose and-goals is based on goal granularity (given at compile-time). Nothing is reported about how to decide between and-work and or-work, and no results are reported at all.

The latest systems that aim at exploiting and-parallelism and or-parallelism and are being implemented are ACE [59], ParAKL [93], PBA [63] and Penny [91]. Some results of a study of and-scheduling strategies for the ParAKL system are reported by Serr [107]. Results for ACE have not been published yet.

To the best of my knowledge, Andorra-I is the first and/or parallel logic programming system that has reported results running real applications. Andorra-I is the first reasonably mature implementation of a system that exploits both and- and or-parallelism. Andorra-I exploits or-parallelism as in Aurora by using the binding arrays technique of the SRI model to solve the problem of different conditional bindings per branch of the search tree. It exploits a simple form of and-parallelism by allowing only *determinate* goals to be executed in parallel. If goals try to bind a common variable with different values the whole branch fails.

A processing element in Andorra-I is called a worker. Workers are forked in the beginning of the execution, and each worker is associated with a processor in the parallel system. This is different from what is done in other logic programming systems, as the system does not allow to fork more processes than there are processors available. This is to avoid overheads of context switching, and interruption of busy workers. Workers in Andorra-I work much the same as in the Aurora system, but instead of having a single worker to explore a branch, several workers cooperate with each other in a *team* in order to explore and-parallelism if this exists in the branch. The first version of Andorra-I used a fixed configuration of workers into teams, and once the user started the execution with a given configuration, this would never change. If the application had a tree that did not suit the chosen configuration, Andorra-I would lose parallelism.

The novel contribution of this thesis is to add to Andorra-I a reconfigurer whose objective is to rearrange workers into teams according to the parallelism existing in the application. This strategy allows workers to be redeployed from one team to another to look for work. It also decides **what** kind of work to choose – and-work or or-work – in order to achieve reasonable load balancing and overall better performance. The other new issue in parallel logic programming systems, either and-parallel or or-parallel, that is tackled by the reconfigurer is the efficient utilisation of resources by the parallel system.

The initial decision algorithm was work-guided, i.e., idle workers would decide to take and-work or or-work depending on the amount of work currently available in the system. This strategy proved to be reasonably good as the results demonstrate. However the amount of work available in the system is not always a qualitative representation of the **actual** amount of work to be done. A scheduler based on such a measure would be good if we had some information on the granularity size of the program, possibly generated through compile-time analysis. Therefore a more reasonable strategy is to allow workers to move freely whenever they are becoming *inefficient* in a team.

We dedicate the next chapter to a thorough description of the Andorra Model and the Andorra-I system, since this is the target system to which we made our main contribution.

4.4 Discussion

Systems that exploit or-parallelism are quite mature and their scheduling techniques are improving with each new algorithm and implementation. Researchers have vastly studied or-scheduler systems and it is becoming clear what is required of an or-parallel scheduler to make an or-parallel system efficient. General conclusions are that low communication between workers is important, in order to be able to perform well with both speculative applications and non-speculative applications. We also need to have an efficient way of keeping the left-to-right order of the branches in the tree and have an efficient algorithm for searching for deepest leftmost work. It is also clear that an or-parallel system can benefit from schedulers that are capable of disposing some processors if the application does not need them. The Bristol-scheduler, the Dharma-scheduler and the Muse-scheduler have been successful in doing their jobs.

Other or-scheduling methods such as the method proposed by Lin [85] are valid, although some assumptions are made that are not easy to accept in logic programming. E.g., structural imbalance is an intrinsic characteristic of logic programs, therefore to remove structural imbalance is not an easy task. To do that at runtime can introduce too many overheads. This strategy is hence more suitable for a static scheduling model than for a dynamic scheduling model. The efficiency of such systems is questionable, because the scheduler needs to be centralised in order to keep track of number of idle processors and number of tasks not yet taken at a given time of the execution, and allocate all idle processors to the work at once.

The and-schedulers are not so studied as the or-schedulers, but some conclusions can be drawn. It is not a very good idea to maintain a global pool of goals to schedule and-goals. Moreover, in systems that exploit IAP it is also important to prevent the trapped goal problem.

The method used by Tick [124], that introduced a compile-time technique to help decisions at runtime, although very simple, proved to be not very efficient for scheduling and-parallelism. It might be useful for scheduling or-parallelism though, as Wai-Keong [48] tried to show. To collect granularity size of programs can be a good idea if one can spot points in the program that present a regular behaviour. This would help the schedulers, both and-schedulers and or-schedulers, in some cases. In the cases that we cannot conclude anything about the shape of the execution tree, some strategy that does not rely on compile-time analysis data needs to be used anyway. To assume that the amount of work in the applications follows a given probabilistic distribution seems unrealistic, given that the shape of Prolog application search trees varies widely. The idea of profiling is interesting and can be a source of future research, although we can profile and draw conclusions for some classes of applications, but not generalise for all applications.

With respect to scheduling both and-parallelism and or-parallelism, as we observed from the previous section, Andorra-I [147] is the only and/or parallel logic programming system that tackles this problem. So far the results have been quite encouraging. Andorra-I is capable of running a wide range of applications that contain both and-parallelism and or-parallelism or either kind alone and achieve good speedups without the need for manually adjusting the number of workers to exploit and- and or-parallelism.

4.5 Summary

In this chapter we presented the concept of *scheduling* in parallel logic programming systems. We discussed about schedulers for or-work and schedulers for and-work. We concentrated first on the most well known schedulers for or-work: the Muse scheduler, the Bristol scheduler, and the Dharma scheduler. Second we gave an overview of schedulers for and-work by describing how to select goals in committed-choice systems, in Andorra-I, and in &-Prolog. Currently, there is no system that employs scheduling strategies for distributing both and-work and or-work, except the ones implemented to Andorra-I. There are one or two proposals of schedulers to distribute both and-work and or-work, but they were not implemented and have some drawbacks. The PEPSys system that exploits and-parallelism and or-parallelism has independent schedulers for and-work and or-work, but does not deal with the problem of exploiting both kinds of parallelism.

Lastly, we gave a discussion about the scheduling models and scheduling systems presented, giving emphasis to scheduling in Andorra-I, which is the target system in this thesis.

Chapter 5

The Andorra Model and Andorra-I

As was mentioned in the two previous chapters, there are two main sources of parallelism in logic programming, namely or-parallelism and and-parallelism. Or-parallelism is exploited when several alternative clauses to a goal are executed in parallel. And-parallelism is exploited when we execute two or more goals simultaneously.

As discussed in chapter 3, there are several ways of combining and-parallelism and or-parallelism in parallel logic programming. One very interesting way of exploiting and-parallelism in logic programs is to execute first goals that match **at most** one clause in the database. The eager execution of determinate goals helps to reduce the search space, because unnecessary choicepoints in the search tree are automatically eliminated. As we discussed before, this kind of combination is used in the Basic Andorra Model and implemented in the Andorra-I system. In general, the combination of or-parallelism with and-parallelism in Andorra-I produces very good results, as we shall see later in this thesis.

In this chapter we discuss the Basic Andorra Model whose main design goal is to exploit and-parallelism and or-parallelism by executing **determinate goals** first. The execution of programs according to the Basic Andorra Model generates *Andorra-trees*. An Andorra-tree is defined as being an or-tree whose arcs, instead of only representing an ancestor relation, represent a graph that correspond to the determinate goals that can be executed in parallel, the *and-graph*. The arcs in the and-graph determine also an ancestor relation as in the or-tree, but can also represent co-operating relations between goals. The Andorra-trees were first defined by Falaschi et al [46], where their description of an Andorra-tree defines the arcs of the or-tree as *Andorra-and trees*. This is not quite correct, because the execution of the determinate and-goals does not generate a tree, due to possible co-routining and stream-and parallelism between goals. So, our definition of Andorra-tree is an or-tree with and-graphs between nodes of the or-tree. We show an example of an Andorra-tree for a program example in the next section.

Also in this chapter, we explain the Andorra mechanism of execution by means of an example and describe an implementation: the Andorra-I system that existed before our work.

5.1 The Basic Andorra Model: A Parallel Execution Model for Logic Programs

In order to illustrate the Andorra computation we give a simple example. Consider the program 5.1, a simple database of facts and rules, that defines several 'papers' (adapted from Conery [26], pp. 10). A paper is defined to have a main subject (e.g., `pprom`: parallel Prolog models, `ai`: abstract interpretation etc), a date of publication, a location, i.e., where it was published, and a title.

Assume now that we enquire for `?- paper(P,1990,bristol,T)`, i.e., all papers published in Bristol in 1990. Initially, a choicepoint is created for `paper/4`. The first clause is tried and, unlike standard Prolog, Andorra-I does not choose the leftmost goal of the first clause for predicate `date/2` to be reduced first, because it is non-determinate (`date(P,D)` has three possible alternatives). Likewise, goal `author/2` is not chosen, because it has several possible alternatives. Instead, goal `loc(A,bristol,1990)` is chosen to be executed first (because it matches at most one clause in the database). The result of the reduction of this goal is to instantiate the variable `A` to `warren`. At this point, the goal `author(P,warren)` becomes determinate and instantiates the variable `P` to `pprom`. When `P` is instantiated, **two** goals become determinate: `date(P,D)` and `title(P,T)`. The Andorra-tree for the execution of the query is shown in figure 5.1.

By using the Andorra mechanism of resolution the first solution for the query is obtained in only 5 steps, whereas standard Prolog would take 8 steps. The difference in reduction steps would be more if we exchanged the fact `date(pprom,1990)` with `date(ai,1990)` in the database, because in this case Prolog would try first to find a solution with subject `ai`, while Andorra would repeat the same execution described. The execution of this program also takes fewer reduction steps under the Basic Andorra execution model than under Conery's and/or process model, because Conery's literal ordering algorithm does not check for determinacy in order to generate the ordered literal graph.

It is important to note that, ideally, any set of determinate goals can be reduced in and-parallel, as long as we have enough resources in the system (this might not be a very practical choice for this example though, because the and-goals of this example do not generate sufficient and-parallel work to justify parallel execution). In addition, the three

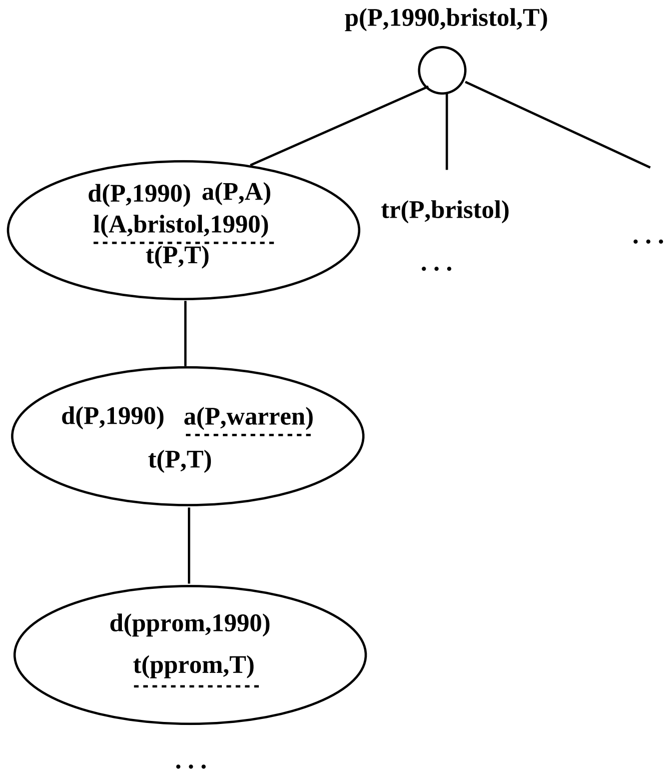


Figure 5.1: ANDORRA-TREE EXAMPLE

alternatives for the query `paper` can be started independently in or-parallel, as long as there are sufficient processors in the system.

The Andorra mechanism of execution allows for a natural form of co-routining, because determinate goals that have structures as arguments can pass incomplete solutions to the next determinate goals that need to consume the structure. In this way Andorra implicitly models a producer and consumer algorithmic relationship between goals.

Andorra also provides a new and powerful programming paradigm. Besides being able to run Prolog programs, Andorra can run committed-choice style programs. Note that in order to guarantee that Prolog programs run correctly in this model one needs to add some kind of preprocessing or control mechanism to avoid goals that cause side-effects being executed out of order, and to avoid solutions being produced out of order.

In the Andorra-I system, a realisation of the Basic Andorra model, correctness is achieved through a preprocessor that guarantees the correct execution of built-in predicates during the determinate phase, and to the or-scheduler that guarantees the correct order of solutions to be output. The user can also force goal sequencing by using the operator '&', supported by the Andorra-I engine.

An important characteristic of the Basic Andorra Model is that parallelism is exploited implicitly and the programmer does not need to use compulsorily any kind of construct to guide parallelism. Moreover Andorra produces all solutions for a query, instead of waiting for the user to enquire for another one. If the user wants only one solution must use a cut operator at the top-level query.

In the next section we describe the Andorra-I system that implements the Basic Andorra Model.

5.2 The Andorra-I System

Andorra-I was implemented in the context of the ESPRIT project PEPMA (Prolog Execution on Parallel Machine Architectures), whose objective was to implement parallel Prolog systems.

As explained before, the Andorra-I system [148, 101] executes **determinate** goals first and concurrently. It exploits and-parallelism in a way similar to the JAM abstract machine [32]. It exploits or-parallelism in much the same way as in the Aurora system [88], by maintaining binding arrays for each worker. Andorra-I exhaustively executes all available determinate goals in the pool of goals until a goal fails or there are no more determinate

```
paper(P,D,I,T) :- date(P,D), author(P,A), loc(A,I,D), title(P,T).
paper(P,D,I,T) :- tr(P,I), date(P,D).
paper(xform,1990,bristol,T) :- tr(xform,T).

author(fp,backus).                  date(fp,1978).
author(ppro,khayri).                date(ppro,1990).
author(pprom,warren).               date(pprom,1990).
author(ppro,yang).                  date(ppro,1991).
author(orsch,beaumont).             date(orsch,1991).
author(ai,costa).                   date(ai,1991).
author(ai,herme).                   date(ai,1990).
author(sasl,turner).                date(sasl,1979).
author(xform,stonebridge).

title(ppro,muse_prolog_system_and_performance).
title(pprom,aurora_parallel_prolog_model).
title(fp,can_programming_be_liberated).
title(ppro,implementation_basic_andorra_model).
title(sasl,a_new_implementation_technique).
title(ai,compile_time_analysis_andorraI).
title(ai,abstract_interpretation_and_applications).
title(orsch,flexible_scheduling_in_aurora).
title(xform,bristol_tech_reports_catalog).

loc(khayri,sics,1990).              journal(fp,cacm).
loc(backus,ibm,1978).               journal(sasl,spe).
loc(yang,bristol,1991).
loc(beaumont,bristol,1991).         tr(db,edinburgh).
loc(costa,bristol,1991).
loc(herme,madrid,1990).
loc(turner,kent,1981).
loc(warren,sri,1982).
loc(warren,bristol,1990).
```

Program 5.1: DATABASE PROGRAM (2).

goals. At this time, the system chooses the leftmost non-determinate goal to create a choicepoint. Thus forward execution is divided into two distinct phases: determinate and non-determinate. During the determinate phase, processing elements are co-operating to execute and-parallel work and during non-determinate phase they are creating new choicepoints to prepare to a new determinate phase. When a goal fails, Andorra-I abandons the failed branch and backtracks to find another alternative not yet executed.

A processing element that performs computation in the system is called a *worker*. Andorra-I is designed in such a way that workers are divided into *teams*. Each team has a *master*, which is responsible for creating new choicepoints and *slaves* that are managed by their master. Workers in a team cooperate with each other in order to share available and-work. Different teams of workers cooperate to exploit or-work. Configuring workers into teams is the most natural way of implementing Andorra-I, given that workers in a branch have to share the stacks in order to exploit and-parallelism, and have to share the BA's with their masters, and other workers in the team.

The master and slave protocol approach is largely used in distributed and parallel systems to implement synchronisation between processors in asynchronous hardware systems. Although this approach centralises the execution on the master, heavier communication protocols necessary for other approaches (e.g., totally distributed algorithms) are avoided.

The system is composed of a preprocessor, responsible for generating determinacy code, the compiled code for the VRAM abstract machine, and the sequencing information necessary to maintain the correct execution of Prolog programs [102, 101, 28]; an engine [148, 147], responsible for the execution of the Andorra-I programs [145, 146, 103]; and a scheduler, that is divided into two components according to their function: to find and-work (and-scheduler), and to find or-work (or-scheduler).

The Andorra-I old design ran with fixed configuration of workers into teams, which means that workers needed to be manually chosen to compose a team.

Figure 5.2 shows the architecture of the system. We have a set of engines (workers) divided into teams, each team working in a branch of the or-tree. The nodes of the or-tree are private or public. Public work is shown in the centre of the picture. Local work is associated with each team and consists of the private part of the or-tree plus and-goals in the run queues of each worker in the team. Workers in a team (masters and slaves) cooperate with each other to carry on their work either by executing goals of their own run queues or by executing goals of other colleagues run queues. Each team has a master worker which is responsible to create new choicepoints and to keep control over its slave workers. Each worker has its own scheduler, which means that scheduling in Andorra-I is distributed and not centralised.

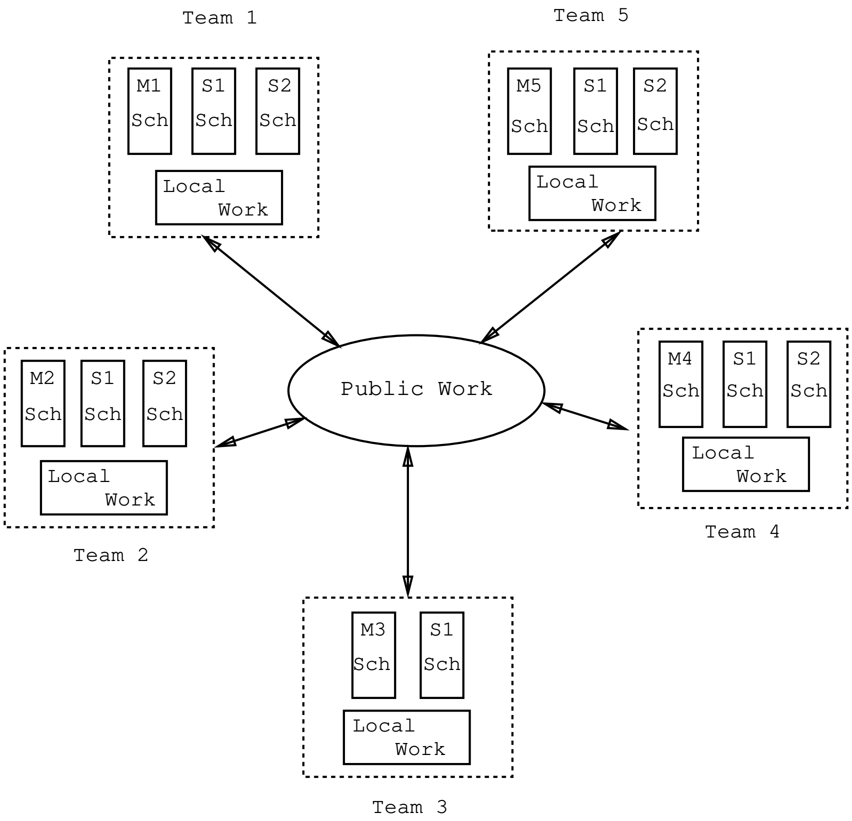


Figure 5.2: ANDORRA-I SYSTEM ARCHITECTURE

In the next sections we will concentrate on the Andorra-I main data structures to support parallelism, the preprocessor, the Andorra-I engine, main implementation features, memory management and scheduling issues.

5.2.1 Data Structures

The Andorra-I implementation relies on the already successful techniques used in sequential Prolog systems to execute Prolog programs. It is a WAM-based implementation, but uses goal-stacking instead of environments. The main reason for that is that goal stacking is more suitable for implementation of and-parallelism, as workers can access the goal-stack easily. The heap and the trail have the same purpose as in the WAM. These structures are shared by workers in a team, but each worker has its own chunk to work on. When a slave's chunk ends, the slave takes another chunk from its master. In order to support co-routining and suspension, Andorra-I associates with variables a suspended goal list stored in the heap. In order to support or-parallelism, the system uses binding arrays [138] inherited from the Aurora system, where each worker records its own bindings to the conditional variables. Therefore, different branches of the search tree have their own independent bindings annotated on their Binding Arrays (BA's). In order to support and-parallelism, Andorra-I uses a machinery similar to JAM Parlog [32] where a circular run queue data structure is employed with private and public parts. Each worker has its own run queue. Each run queue has 2 pointers. The owner worker pushes onto the rightmost pointer and a remote worker removes a goal from the leftmost pointer. The left-to-right order of goals, necessary to find the leftmost non-determinate goal when the determinate phase finishes, is maintained by a double-linked chain that is implemented with difference lists.

The worker data structure is consulted whenever running in parallel. It contains information about the status of the worker (runnable, idle, busy etc), its type (master or slave), its membership list, and other fields that are important for memory management, backtracking and pruning operations.

5.2.2 Memory Management

Memory management in Andorra-I is not very optimised at the moment. The workers are responsible for dynamically allocating a fixed amount of memory for the stacks, binding arrays and other data areas, but this amount does not change dynamically during execution. There is no form of garbage collection either. Workers in a team have their stack memory shared among them and each slave can allocate chunks of memory from the available memory managed by its master. The chunk size can significantly affect the system's performance and if well chosen allows for reasonable memory savings and better performance.

5.2.3 Preprocessing

Andorra-I does not execute Prolog programs as such. In order to maintain correctness of sequential programs, deal with side-effects, pruning operators and detect determinacy, the programs are transformed by a preprocessor [102]. The Andorra-I preprocessor, besides generating determinacy and compiled code with a very sophisticated indexing mechanism, can use techniques of abstract interpretation to infer mode and determinacy information. It generates code for goals that are always proved to be determinate, always non-determinate and, when nothing can be decided, it generates a decision tree that is tested at runtime to detect determinacy of goals. It also generates Andorra-I compiled code (VRAM instructions) based on the JAM abstract machine [32]. The Andorra-I engine executes preprocessed Prolog programs.

5.2.4 Execution Mechanism

The Andorra-I engine worker cycles are depicted in Figures 5.3 and 5.4 (pictures adapted from Yang [143]).

A slave can be in any of the following states:

- *Busy* when it is performing any kind of reduction (determinacy tests or Prolog reductions).
- *Runnable* when it grabbed a goal from some worker's run queue and wants to guarantee that this goal will not be stolen by another worker.
- *Idle* when its run queue is empty.
- *Failed* when it detected by itself or from other workers that its branch failed.
- *Undo-binding* when it fails it needs to undo its bindings on its stacks' chunks.
- *Failure-Waiting* after undo bindings the worker completed backtracking and waits for more work to be created by its master or other slaves in another branch.
- *Idle-Waiting* when is idle and waiting for some communication with its master or from its colleagues of the same team.

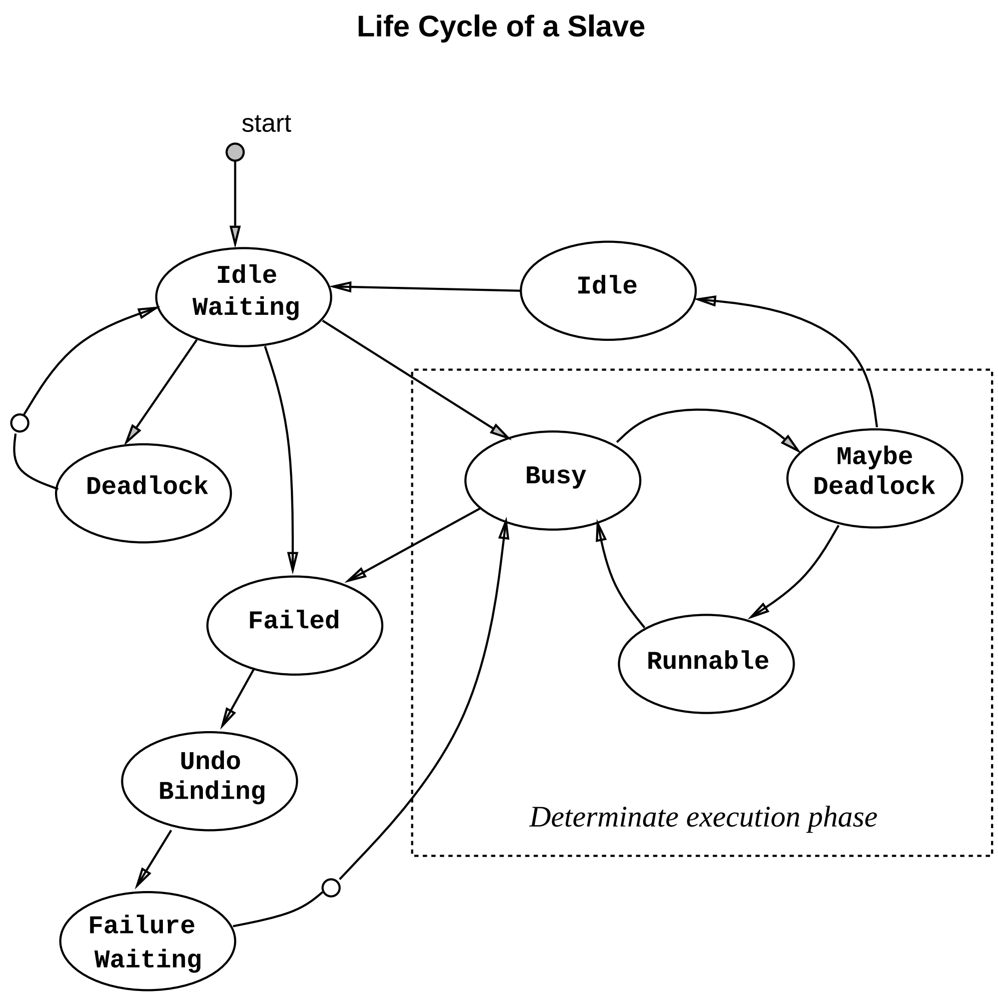


Figure 5.3: ANDORRA-I SLAVE ENGINE CYCLE

- *Maybe-Deadlock* it enters this state whenever it does not find any work in its team. Stays in this state until some other worker wakes it up to inform it that there is more work available or until a deadlock situation happens.

- *Deadlock* this is a situation when there is no more work to be done, because either the branch failed or the determinate phase finished. Andorra-I breaks this deadlock by allowing the master to choose the leftmost goal in the run queue of goals and enter the non-determinate phase.

A slave starts idle and enters the determinate execution phase if there is any run queue in its team with goals that can be executed in parallel. Once the slave picks up a goal from a run queue it becomes runnable. This state was created only to avoid that the worker receives a request to pick up the same goal just taken, before it has a chance to start its own reduction. Once it starts reductions, the worker becomes busy. If its branch failed, the slave undoes its bindings and enters in a failure waiting state until it becomes busy again. If the slave is still idle after the determinate phase finishes, it enters a deadlock state until its master breaks the deadlock by creating a choicepoint.

A master has a slightly different engine cycle, because it needs to coordinate its slaves and create the choicepoints. Inside the determinate phase both workers (masters and slaves) behave in much the same way, but when the determinate phase finishes or the master becomes idle, a situation of deadlock might be reached. If the master concludes that the system is in deadlock, it treats the deadlock by sending a signal to its slaves. It enters the non-determinate phase by making a choicepoint. If its branch failed, it undoes its bindings and starts backtracking until it gets busy again.

Next we present in more detail the forward and backward mechanisms of execution in Andorra-I and how parallelism is managed by the system. Some names given below are according to the standard interface used by the engine and its schedulers [120]. Engine interface macros mentioned in the next sections have no prefix. Or-scheduler functions have `Sched` prefix. And-schedulers functions have `And_Sched` prefix.

The old Andorra-I with fixed configuration required that the user chose the number of workers (masters and slaves) to be used by the application. Therefore, the user needed to start the system by giving in the line command: `-M m -S s`, where `-M m` gives the number of masters `m`, and `-S s` gives the number of slaves `s`. This means that the slaves will be evenly distributed among the masters forming $m - mod(s/m)$ teams of s/m slaves and $mod(s/m)$ teams with $s/m + 1$ slaves.

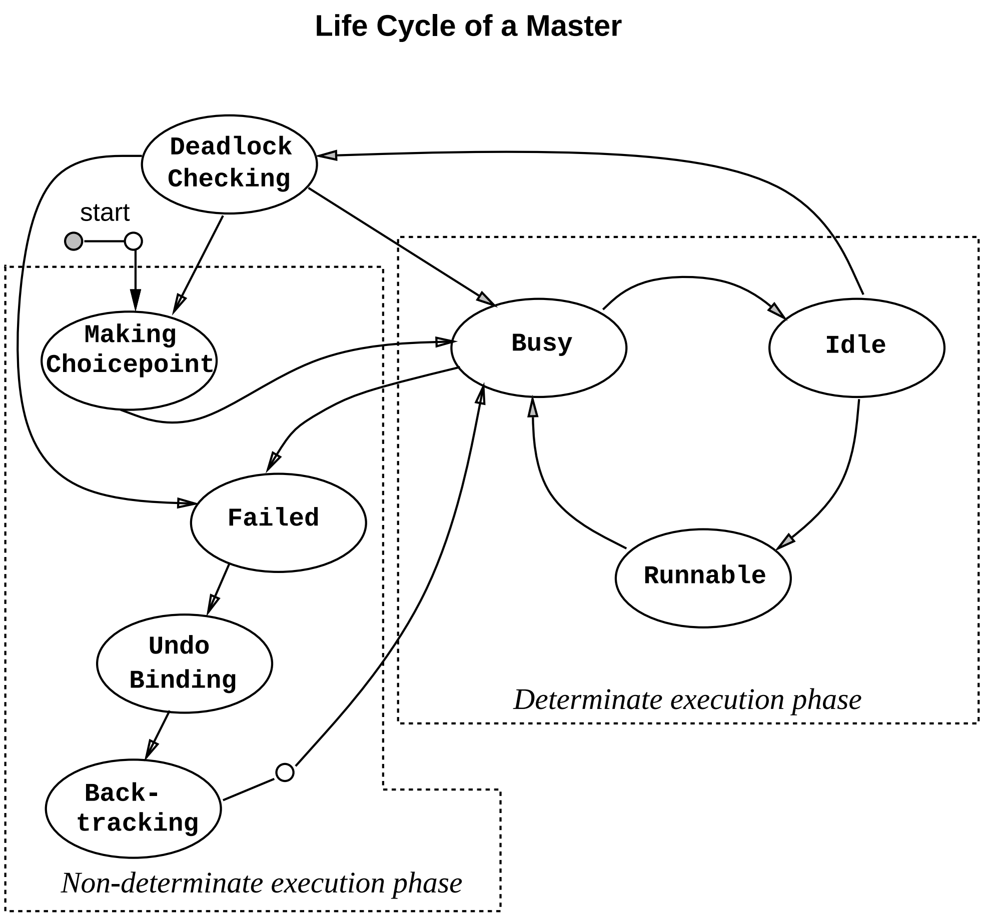


Figure 5.4: ANDORRA-I MASTER ENGINE CYCLE

Forward Execution When the system starts, before forking, the first worker arranges the fixed configuration by creating as many teams as the number of masters and distributing the slaves evenly among the teams. For example, if the user gives as parameters `-M 3 -S 7`, three teams will be created, the first team with three workers and the other two teams with 2 workers in each.

The first process in the system (called **leader**) is responsible for creating $\mathtt{n} - 1$ workers by forking, where n is the total number of processors ($\mathtt{m} + \mathtt{s}$) chosen to run the application.

After forking, each worker calls the macro **Sched_Set_Up_Worker**, in order to do the necessary or-scheduler initialisations. The master of each team calls **Sched_Start_Work** to find or-work. Notice that this master would be idle until some or-work appears during the computation. If no or-work appears this worker is under-utilised.

Each slave will try to take new work from its own local queue. If it is empty, the slave worker will call the **And_Sched_Find_Work** macro to try to find some work on run queues of other workers of its team. If some work is found, the worker is marked as Runnable, the goal is removed from the queue (after locking), and the run queue is updated. If no work is found, the slave enters a Maybe-Deadlock status and will be woken up, either if some and-work is created in its team, or if the determinate phase finishes for any reason. Each run queue must have information about whether it is empty or not empty. This information is used by the and-scheduler.

Once a slave does not find any work in its team, it will loop idle till the end of the computation or until some other worker creates more and-work in its team.

Initially, the slaves' local queues are empty and the work will probably be found in the run queue of their master. At this time, only the leader master worker has some work to do. The leader can create the first choicepoint from the only alternative from the root node and can start the determinate or non-determinate execution.

A master worker is responsible to create choicepoints when all the run queues in a team are empty or when some situation of deadlock happens. After creating a choicepoint, it starts again the determinate phase by taking some goal from its own run queue.

Backward Execution When a fail occurs, the engine that first found the fail signals its colleagues in the team and waits for everyone to acknowledge the failure signal. All the workers check, from time to time, if a fail occurred in their branch. When the master detects that a fail occurred it waits for their slaves. After realising that all the workers stopped, the master can start its backtracking phase, moving to the nearest choicepoint (nearest long life node) and position there looking for new work.

The master will first try a **private** backtracking. If there are no more alternatives to be taken from a private node of the tree, the worker calls the or-scheduler that will be responsible for doing a **public** backtracking and looking for or-work. At this moment some nodes can be made public to other workers.

It is important to note that, in Andorra-I, once a fail occurs in a branch, the whole computation on that branch fails because the failed goal was determinate. There are no choicepoints made in a determinate phase.

Handling Suspensions, Cuts and Commits During execution, a worker can be doing one of the following:

1. executing a normal goal;
2. giving up a goal because it has a suspended variable;
3. executing a side-effect predicate;
4. executing a cut or commit operator;
5. being stopped by a pruning operation;
6. being stopped by a fail in its branch;
7. voluntarily suspending work.

The six cases will result, sooner or later, in a call to the scheduler.

The first, fourth, fifth and sixth cases are those directly related to communication among workers.

In the first case, the worker carries on its work until the computation on its branch fails, or until the computation fails, or until it is pruned, or until it is interrupted by the user or by the system, or until the worker itself tries voluntary suspension. While executing work, a worker calls the **Sched_Check** macro to verify if there are any workers requesting or-work from it or to make some nodes public. Eventually, the goal being executed can fail and this will cause all the other workers to be in the sixth condition. It can also happen that a worker is executing a normal goal and is interrupted by some other worker that executed a pruning operation.

In the second case, the worker simply gives up the work, puts the goal in a suspended goal list (e.g. waiting for a variable) and carries on with another goal in its run queue.

As third condition, the engine is trying to execute a side-effect predicate. When it is being executed inside a conjunction, the determinacy code is responsible for keeping the leftmost order execution of side-effect goals by causing their suspension until they become leftmost in the conjunction. Whenever a side-effect goal is about to be executed, then the worker calls the or-scheduler **Sched_Synch** macro to know if its branch is leftmost in the whole tree. If it is the case that the branch is not leftmost, then a code is executed to suspend the branch (the worker calls the **Sched_Suspend** function after doing the necessary housekeeping operations). The worker might find work elsewhere in the tree and start a new branch.

To the fourth case, the worker calls the **Sched_Prune** macro that is responsible to inform its right sibling workers that their branches are being pruned.

In the fifth case, the worker is being informed that it has to prune its work. All master workers check if their branches were pruned by another worker of other team that executed a cut or commit. This is done by calling the **Sched_Be_Pruned** macro. The slaves also check if their branch was pruned. They are informed by their masters about this condition. The master must wait for the slaves to stop in order to start backtracking.

In the case a fail occurs, the worker is being informed that a fail occurred. This proceeding was explained before. All the workers in the same team call the and-scheduler to check if their branch failed.

In case of suspension, the code for suspension is valid only for the or-scheduler interface. For and-execution, the suspension is treated by the engine, not by the scheduler.

Each engine is always checking if it has to stop working for any of these three reasons: interruption, pruning or failure.

Handling Interruption of other Workers As mentioned before, there are three situations when workers have to communicate with each other: failure, pruning and requesting work.

During the determinate phase the workers communicate with each other by a flag-field in the worker structure that indicates if the and-execution failed. In case of executing a cut, the worker does not need to communicate with other workers in the same team, because the determinacy code ensures that determinate goals that comes after the cut are not executed before the cut has been executed.

Master workers should communicate with other workers in case of pruning. If one of the or-branches is executing a cut, it has to set a variable saying that its right neighbours should be pruned. The same happens if the worker is executing a commit, but in this case the worker has to inform also its left neighbours about the pruning.

Master workers also communicate with other master workers in case of a request for or-work. The requesting master sets a field at the other master's data structure to indicate it is interested in or-work. The other master will check this field from time to time while working.

5.2.5 Constraint Logic Programming in Andorra-I

In addition to supporting committed-choice style languages, Andorra-I also supports parallel and concurrent constraint programming. The constraint domain predicates and operators implemented are similar to the ones defined in CHIP [39]. Andorra-I thus provides parallel execution of constraints exploiting both and- and or-parallelism. The speedups are quite good for both and- and or-parallelism, proving that constraint programming can also be useful in the scope of and-parallelism [53]. One of the benchmarks used in this thesis, presented in chapter 8, benefits from the use of constraints.

5.3 Scheduling in Andorra-I

The schedulers in Andorra-I are designed with global and local parts (or private and public in the case of the or-scheduler) in order that we do not concentrate all the decisions on just one scheduler block. There is no global dispatcher or dedicated process to do scheduling. Each worker is responsible for its own scheduling. With respect to the and-scheduler, it is a good policy to keep different pools of goals (run queues) in order that workers do not generate contention to access the work available. Likewise it is a good policy to have the tree divided into regions in order that the search for or-work does not dominate the execution time. When some worker runs out of work in its local pool, it calls the reconfigurer to distribute and- or or-work.

5.3.1 The And-Scheduler

The and-scheduling implemented in Andorra-I uses global queues with private parts. This is shown to produce good balance between contention and exploitation of and-parallelism, although a mechanism to push goals on the public queue is used to restrict workers from keeping selfishly all goals for themselves. Goals are stored in a run queue of goals in a stacked manner. The global queues contain 2 pointers. The owner of the queue pushes goals from one side and the remote worker takes a goal from the other side. Private queues are only used by their owner worker.

Once a slave scans the run queues of its colleagues and finds out that there is no more and-work to be done, it will enter a "Maybe-Deadlock" status. The slave will not try to find more work until some worker "wakes it up" by sending a signal different from Maybe-Deadlock.

One special situation needs to be handled: a worker needs to guarantee that it will take one goal from the run queue and this will be executed by itself. This is achieved by the Runnable status. While a worker is in a runnable status, other slaves will give up looking at its run queue. This solves the problem of "one stealing from the other and the other stealing again from the one".

The current status of the and-scheduler in Andorra-I is such that goals are scheduled from left to right, following the order (Prolog order) of the pool of goals. This pool of goals is implemented as a sideway-linked chain [148], inspired by Prolog difference lists, to allow quick access to the leftmost goal during the non-determinate phase.

Currently there are some features to stop selecting a goal depending on its depth, and a scheme to change the order of pushing and removing goals from the queues. This can avoid generation of infinite branches. The depth parameter is also used to force a worker to push goals always on its own run queue. This is done to avoid too many suspensions and lockings when the and-granularity is fine. No information obtained from static analysis is utilised to selecting goals, therefore, at the moment, the user needs to tune the depth bound in order to achieve better performance for some programs.

When exploiting dependent and-parallelism, Andorra-I uses synchronisation to execute goals in the context of cut and commit operators, and side-effect built-in predicates (it is up to the determinacy preprocessor to generate this kind of synchronisation).

5.3.2 The Or-Scheduler

The or-scheduler that is currently being used by Andorra-I is the Bristol Scheduler [7]. Its main function is to find the best pieces of or-work in the tree, by using a hybrid strategy. While working in mandatory work it uses the strategy *start random, finish work*. This means that the worker will start looking for work from any randomly chosen node in the tree and, once the alternative is taken, it is run to completion. If the worker is working on any part of the tree considered to be speculative (e.g., a worker is working in the scope of a cut or commit), it considers to do a *voluntary* suspension and look for other better work. This strategy is known as *start left, seek for better work*, because the worker will start to look for work from the leftmost branch (heuristically, the less speculative task in the execution tree) and periodically suspend to seek for better work.

Another issue tackled by the Bristol scheduler is the ability to use only as many resources as the system needs. This is a very important issue when we are using a multi-user system environment. In order to take or dispose resources to the system, the Bristol scheduler monitors the worker efficiency, i.e., the scheduler keeps a record of the *idleness* of the worker measured since the worker is woken up. From the moment it wakes up, the worker starts to record the time that is busy. If while busy, the efficiency %busytime - %idletime drops below a threshold, the worker goes to sleep. If while idle, the efficiency is below a threshold, the worker goes unconditionally to sleep.

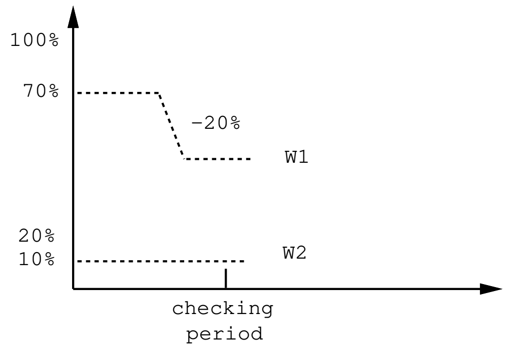


Figure 5.5: EXAMPLE OF EFFICIENCY CONTROL IN THE BRISTOL SCHEDULER

Figure 5.5 shows an example of monitoring efficiency. For example, if a worker is working with efficiency 70% (%busytime - %idle-time = 70%) and this efficiency drops to 20%, then the worker goes to sleep. If a worker is working with 10% of efficiency, it goes to sleep unconditionally. Figure 5.5 shows graphically the example.

5.4 Discussion

As we can conclude from this chapter, there is scope for many improvements to the way the Andorra-I system was originally designed. The most important ones directly related to our work are:

- We need to remove from the user the burden of having to choose fixed configurations of teams in order to run a particular application, because this goes against the main objective of Andorra-I, that is to exploit parallelism implicitly.
- There are many points in the master and slave cycles where the workers are completely under utilised and stay looping idle, because there is not enough parallelism in the team where they are allocated.

In the next chapter, we will discuss issues and alternative ways of distributing workers *automatically* in Andorra-I.

5.5 Summary

This chapter had as an objective to give a detailed presentation of the *Basic Andorra Model* and the *Andorra-I* system, a realisation of the Basic Andorra Model.

We first described the Andorra computational model by means of a program example. Next we described the Andorra-I system and its main components: the engine, the preprocessor, and the schedulers. The chapter gave an overview of the system's implementation through its architecture and through life cycles of *masters* and *slaves*. We explained the forward, backward and synchronisation algorithms, and the systems' interface with the or-scheduler and with the and-scheduler. We also showed how the system handles suspensions, cuts, commits, and interruption of other workers in the parallel version.

Lastly, we discussed the problems existing in Andorra-I, namely the form of distributing and- and or-work, where *fixed configuration of workers into teams* is used. We claim that fixed configuration of workers into teams is inconvenient for the user, it goes against the aim of the system that is to exploit parallelism implicitly, and waste resources.

Chapter 6

Reconfiguring in Andorra-I

The scheduling models for parallel logic programming systems presented in chapter 4 use different methods to deal with the problem of distributing only one kind of work among workers, whether it be and-work or or-work. None of them presents a solution to the problem of distributing workers between two different kinds of work.

Scheduling multiple forms of parallelism in parallel logic programming systems is a new subject. Implementors of parallel logic programming systems are still trying to solve problems related to data representation, what kind of parallelism to exploit, implementation methods to combine both and- and or-parallelism etc. No attempt had been made so far to investigate the problem of solving the distribution of and-work and or-work that arise naturally in a parallel logic programming system that exploits several forms of parallelism.

In the last chapter, we described the Andorra-I system in detail. Andorra-I is a system that exploits multiple forms of parallelism. Systems that exploit several forms of parallelism, such as Andorra-I, face a new problem: *how to transparently and efficiently distribute and-work and or-work that arise dynamically and irregularly during the computation of a program*? By *transparent* we mean that the system should exploit parallelism without any user's intervention. By *efficient* we mean to distribute both kinds of work and achieve speedups close to a possible optimal. By *dynamic* and *irregular* we mean that the sizes and forms of parallelism vary with time.

In this chapter, we discuss the main issues in distributing work among processors in Andorra-I, discuss limitations of the Andorra-I system, and propose solutions for the problem of distributing both kinds of work.

6.1 Andorra-I in the past

Andorra-I, as described in the previous chapter, is designed in a way that workers are grouped into teams. Each team is composed of a master and some slaves. Workers in a team cooperate to share available and-parallel work, while teams cooperate with each other to exploit or-parallelism. 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.

The main objective of our work is to add a key missing component to Andorra-I, that will automatically distribute workers between the two kinds of work available, and thus release the user from the burden of having to adjust manually fixed configurations of masters and slaves into teams.

In order to achieve this goal, and hence make Andorra-I a practical system for the user, there are at least three possible approaches. In the first approach we can use compile-time analysis of the programs for a certain set of queries (probably the most common) to generate information about the best fixed configuration of masters and slaves that should be used for that particular computation. In the second approach the work distribution is decided at runtime. In the third approach the work distribution is decided at runtime, but guided by information generated through compile-time analysis.

The first approach has three main disadvantages:

- The fixed configuration is set forever in the beginning of the computation and *never* changes. This may lead to loss of parallelism, since the parallelism varies with execution time.
- The process of collecting information about the amount of parallelism in a program in order to find a suitable fixed configuration of workers into teams is very complicated

and sometimes does not produce the precise and expected results. First because the computation of an application varies greatly with different queries and different sizes of data. Second because some variable dependencies in the program are only solved at runtime, which makes the task of generating *precise* or even *useful* information more difficult.

- Usually, the process of obtaining useful information through compile-time analysis is very slow, therefore the overall gain in running the application in a parallel system may not be justified.

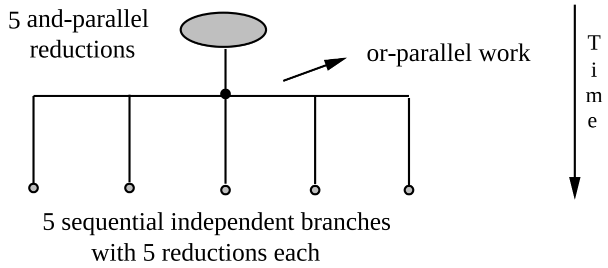
The second approach seems to be better than the first one. Dynamic reconfiguration allows the workers to react to the runtime system, and therefore gives a chance to the workers to change their position according to events that happen at runtime. It has also some disadvantages. The first one is that dynamic strategies, in general, do not find global optima. However, a fixed configuration would not achieve a global optimum either. The second disadvantage is that we have the overhead of re-scheduling at several stages of the computation. But, as long as we keep scheduling overhead costs and frequency of task switching low, this disadvantage is manageable.

The third approach, which is to combine a static schedule with dynamic scheduling seems to be the best choice, as long as the static information can guide the dynamic scheduling decisions whenever possible, and overheads are kept low.

Although we think that the combination of dynamic plus static information is a better choice to guide scheduling decisions, we concentrated only on the dynamic aspect of the problem (leaving scope to static information to be used), as the issue of generating static information for the programs is a whole subject by itself. In any case, as we believe that static information alone is not sufficient to do a good schedule, but totally dynamic scheduling will suffice, we concentrated on the main problem, that is to find a *dynamic* solution for distributing work that varies *dynamically*.

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 (25 workers arranged in 5 teams would do the job!) 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. Figure 6.1 illustrates the problem.

Config	time units
1M4S	26
2M3S	18
3M2S	14
4M1S	14
5M	10



Consider that all branches in this tree do not have any parallel and-work, and the first and-parallel phase has 5-fold speedup. The table shows the number of time units to complete the execution with 5 different fixed configurations of workers into teams. The ideal execution time for this tree at 5 workers is 6 (5 workers in a single team to exploit the and-parallelism in the beginning, and 5 teams to exploit each independent or-parallel branch), which is much faster than any result produced by any fixed configuration shown in the table.

Figure 6.1: AN ANDORRA IRREGULAR TREE

Besides trying to solve the new issue of what kind of work to distribute, the design of the reconfigurer also includes the control of resource utilisation by requesting and disposing workers to the operating system according to the demand of work. This is a very important issue that has not been taken into account by the implementors of parallel logic programming systems, except in Aurora [5]. The motivation for tackling this problem is that, in parallel environments, there is a cost to use resources. Therefore the application should aim at using only the necessary resources. As most parallel systems are multi-user, a parallel application should not be selfish when using the resources available.

In areas other than logic programming, researchers have been studying this problem [125, 150, 129, 33], either by introducing "intelligent" functions into the operating systems [125] to be used by the implementor of the parallel system or by totally leaving the control of resources utilisation to the implementor of the parallel system. The scheduling strategies we propose to use in Andorra-I aim at solving these problems.

6.2 Adding the Reconfigurer to the old Andorra-I

We modified the Andorra-I original implementation in order to accommodate the reconfigurer. The main changes were made to the engine, in order to incorporate calls to the reconfigurer when the determinate phase finishes, to the or-scheduler in order to add calls to the reconfigurer during idle loops or searching for work, and to the synchronisation algorithms to allow workers to enter a determinate phase and help a team. Besides some changes to the Andorra-I algorithms, we also incorporated into the system new functions to interface engine, or-scheduler, and-scheduler, and reconfigurer. These functions are shown in appendix A. The main ones are the functions used by the reconfigurer to obtain runtime information to guide decisions.

The engine calls the reconfigurer in order to initialise its worker and team data structures at the very beginning of the execution, whenever a worker finishes its determinate execution, and every so often to check the status of the execution. From the engine side a locking field needed to be added to the worker data structure in order to allow workers to enter a team one at a time.

The or-scheduler calls the reconfigurer whenever a master is idle for more than a quota of time. This can happen when the master is doing public backtrack to find or-work, at the very beginning of the system, when Andorra-I is started with a configuration of more than one master, and when a master is suspending a branch to look for another piece of or-work.

The reconfigurer assumes that, in Andorra-I, workers can be redeployed in the following way: *from master to slave*, *from slave to master*, and *from slave in one team to slave in another team*.

While a master is idle or backtracking, slaves can be redeployed to another team. A master can only become a slave if all its slaves were redeployed to some other team.

The following is the new algorithm of the engine in Andorra-I, after the dynamic reconfiguring was introduced to the system.

- Master
  - look for or-work;
  - if some work is found return and start the new work;
  - otherwise the or-scheduler calls the reconfigurer;
  - if the reconfigurer decides to change the status of the worker and it becomes slave, the control returns to the engine;
    - a slave cycle starts;
  - if the reconfigurer decides that the worker continues as a master, it returns normally and the engine carries on the execution to find or-work.

- Slave
  - try to find work in its own run queue;
  - if some work is found, keep going;
  - otherwise try to find work in its colleagues' run queues;
  - if some work is found return normally;
  - otherwise call the reconfigurer;
  - if the reconfigurer decides to change the status of the worker and it becomes a master, the control returns to the engine;
    * a master cycle starts;
  - if the reconfigurer decides that the worker continues as a slave in its team or in another team, it returns normally and the engine tries again to find work from its colleagues' run queues.

These modifications to the main Andorra-I algorithms show how the system was changed to support any reconfigurer.

6.3 Strategies Proposed

So far, we have not discussed what strategy to use by the reconfigurer in distributing workers between and-work and or-work. We chose two different approaches to defining a reconfiguring strategy. One approach consists of observing the present state, and possibly indications of the future of the computation, in order to predict what kind of configuration is most suitable. The other approach is to observe the past of the computation and be able to monitor the performance of workers over time. We exploit both approaches in this work. The first approach is what we call the *work-guided* strategy, because it is based on a prediction of amount of work still to be done, and the second approach is what we call the *efficiency-guided* strategy, because it is based on the past efficiency of the worker.

If the strategy for reconfiguring is based on the present/future state of the computation, we can use *estimates* of sizes of and- and or-work as an approximate prediction of the future. Ideally, we would like to have the correct sizes of and-work and or-work in order to make the right decisions most of the time. However, as mentioned before, to predict the correct sizes of and-work and or-work is very difficult, because runs of the programs vary widely depending on the data size and on the input query. If, on the other hand, the strategy is based on the past of the computation, we can use measures of processor performance.

Both strategies use reasonable heuristics. The work-guided strategy will always reconfigure workers according to the estimated amounts of work existing at runtime. If the estimates for the amounts of work are precise enough, the reconfigurer will make right reconfigurations most of the time. The efficiency-guided strategy will always reconfigure workers according to their efficiency measured as a percentage of working time over total execution time. As the aim of every parallel system is to keep processors busy most of the time, this also seems a reasonable heuristic.

A good reconfiguring strategy should achieve the following:

- Never leave a worker idle while there is work available.
- Whenever possible do not redeploy a worker to do some work that takes less time to execute than the cost of redeployment, unless there is nothing better to do.
- Always try to move to the piece of work which has the best ratio between amount of work and cost of redeployment.
- Avoid redeploying a worker to a team that has very fine grained and-parallelism, because in Andorra-I once slaves are redeployed to a team they can only leave when the determinate phase finishes[1], and having more workers than the work available to be done in a team will waste resources.

6.4 Andorra-I Sources of Overheads

In Andorra-I, several factors contribute to add overheads to the system:

- Sharing of and-work among workers in the same team creates contention to access the run queues. In order to share and-parallel work in a team, workers need to synchronise to get access to the run queues of goals. The more workers there are in a team, the more contention we have.
- Slaves need to synchronise with their master in order to link execution environments and start a new computational phase.
- Sharing or-work demands synchronisation and communication among masters of different teams (teams interrupt others to get more work, workers making nodes public, accessing the public part of the tree etc).

[1]This is a restriction on the implementation of Andorra-I. A version of the system that allows slaves to leave during the determinate phase was implemented, but it did not work, yet!

- Moving workers up and down the tree is another source of overhead, because masters need to install and remove bindings in their binding arrays.

Dynamic reconfiguring introduces new overheads to the system.

In practice, we need to take the overheads of the system into account, in order to do appropriate redeployments without adding excessive overhead costs to the system (low reconfiguring overheads), and in order to do reconfiguring that leads to low costs overall (good reconfiguring heuristic). The ideal aim is to have low overheads for the reconfiguring algorithms, and right redeployments whenever possible. Therefore, a knowledge of the various overhead costs of the system serves to help design reconfiguring strategies meeting the criteria mentioned in the last section.

Another reason why we need to measure these overheads is that information about time spent in activities that are not related to performing reductions is a source of study and possible optimisations for the Andorra-I system as a whole.

6.5 Identifying and Measuring Major System Activities

In order to collect overhead costs from Andorra-I we classify the time spent by the system in several activities. This information will also be used later in chapter 10 to study the performance of different strategies.

Each worker in Andorra-I performs two basic activities: (1) it executes actual Prolog code plus determinacy tests; and (2) it performs some scheduling activity. All the remaining time is spent idle-waiting. We shall name the time spent in different activities as:

- (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). This should be approximately the same as the sequential time S, if no extra work is performed related to the work done by the sequential system.
- (I) *idling*: this includes all time spent idle, such as waiting for a signal from a worker and synchronisation between workers.

- (E) *extra work*: time spent doing extra work (speculative work).

We shall also define R as the number of processors available to run an application and P as the parallel time to execute an application.

Ideally, the parallel execution time for any application would be given by:

$$P = \frac{S}{R}$$

where S is the sequential time that should be approximately the same as W, if no extra work is done, and ignoring parallel contention.

Unfortunately, due to the system's overheads and limits to the parallelism available in the application, the parallel execution time is given by:

$$P = \frac{W + O + A + T + I + E}{R}$$

The metric $(O + A + T + I + E)$ is commonly called *lost cycles*. To study the performance of a parallel system by looking at these measures is better than to only look at speedups, because we can isolate several sources of system's overheads that affect the parallel execution of a program, since we know that W will be approximately the same as S.

This kind of performance analysis relates to the work of Crovella [34] about performance debugging in parallel systems. He uses the same metrics to measure sources of overheads, but employs a different instrumentation technique. Our approach in measuring performance is different for we do not allocate one dedicated process to account for timing measurements. Moreover we do not employ *performance predicates* to monitor the parallel execution. Instead we monitor the code by using a performance evaluation tool (the micro-second interface) that is called at certain strategic points in the C code. Although our approach is also a predicate profiling technique, the way we implement it is different from Crovella's. Part of the profiling could be done with `gprof`, but this tool, supposed to generate a profile of parallel applications, does not work for parallel applications on the Sequent machine we are using.

Our concept of *speedup* is according to usual convention: $speedup = \frac{S}{P}$, i.e, sequential time over parallel time. In theory, if we follow the formula for P given before, this would give the following speedup formula:

$$\frac{S \times R}{S + O + A + T + I + E}$$

If the overheads and time to do extra computation are equal zero, we have maximum speedup, otherwise, depending on the overheads, we will have speedup below the maximum ideally achievable. Given that we unavoidably already have $A + O$ overheads, when designing the reconfigurer, we wished to keep T and I as low as possible.

6.6 Integrating the Micro-Second Interface in Andorra-I

The timing and frequency of activities at execution time were collected through the Micro-second interface [118], that is an extension of the one previously used by Szeredi to analyse performance of the Aurora or-parallel system with the Manchester Scheduler [119]. The major goals of the measurement in Aurora were:

- to measure the overheads associated with the scheduling activities;
- to measure the time a worker performs useful work;
- to measure the time a worker remains idle.

We have the same objectives, but in addition we are also interested in:

- individual costs of exploiting and-parallelism and or-parallelism;
- estimating the costs of reconfiguring.

The micro-second interface was originally implemented in Aurora with two goals: (1) to avoid the use of the standard `time()` system call because of its high overhead and distortions caused to the parallel execution, and lack of accuracy; (2) with the software engineering purpose of making a clear and general implementation of instrumentation functions that could be used easily by other systems.

The original micro-second interface was changed in order to make clearer the boundaries among the engine, the or-scheduler, the reconfigurer and the interface itself. The interface was adapted to have its own data structures that are used by the workers via interface macros. Another change was made to allow timing of just query execution. A new flag was introduced to make sure timing is only effectively done during execution of a query. So, a built-in predicate is called before starting the query in order to initialise the flag. Profiled data is stored by an `end-timing` Prolog predicate.

Another change was introduced that is related to the utilisation of `goto` loops and `jumps` used in the Andorra-I engine. An array flag was introduced to the micro-second interface in order to ensure that events are always timed only once and in pairs `begin-timing-end-timing`.

The Andorra-I worker cycle with respect to finding work is shown in Figure 6.2, and includes the reconfigurer actions. Figure 6.3 shows the main timing operations in Andorra-I with respect to the worker cycle.

Figure 6.3 shows just part of the timing algorithm where the workers are transitioning from idle to busy status and vice-versa. Each black dot represents a mark for the end of an instrumentation operation and the beginning of another instrumentation operation.

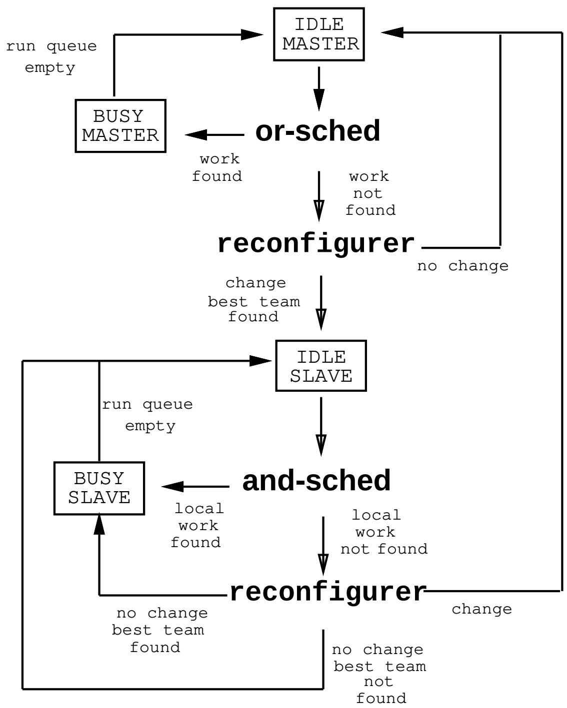


Figure 6.2: LIFE CYCLE OF A WORKER

With the instrumentation of the system we can obtain two sources of information:

- Total accumulated time for all processors for a given activity, whether it be reconfiguring, or-scheduling, and-scheduling, working, or idling.

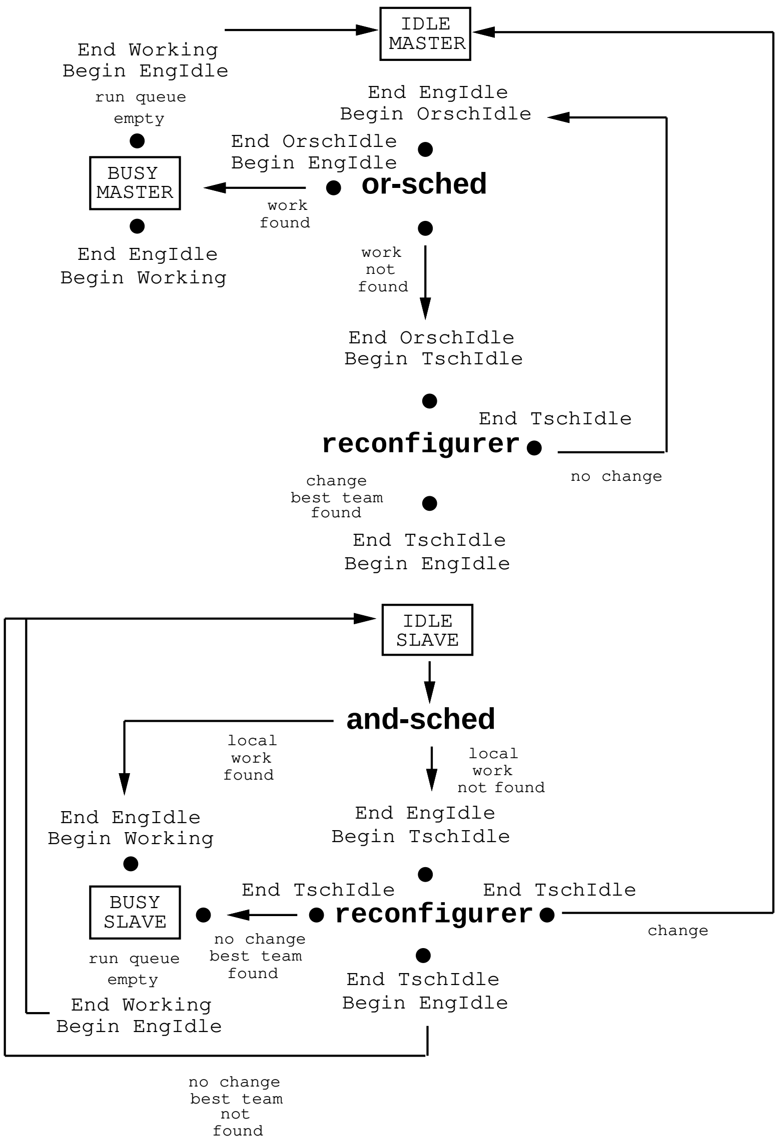


Figure 6.3: TIMING OPERATIONS FOR A WORKER

- The unit cost of each activity in the system, whether it be reconfiguring, and-scheduling, or or-scheduling.

The first source of information will be used later to study two reconfiguring strategies. It is collected by having a counter for each function in the system for each processor, and summing up the total time spent on the main activity. The second source of information is used as a framework for helping the design of the reconfiguring strategies. It is taken as a ratio between the total accumulated time spent on the function during execution and the total accumulated frequency that the function is used in the system. Running a version of the system with reconfiguring, we obtained the information contained in tables 6.1 and 6.2.

Table 6.1 shows the average costs of each function that represents the reconfiguring activity, obtained at 10 processors. Column `M→S` shows the unitary cost of redeploying a worker from a master to slave assuming a particular reconfiguring strategy, that will be described later. The column `S→S` shows the unitary cost of redeploying a worker from one team to another. The last column `S→M` shows the unitary cost of redeploying a worker from a slave to master. The costs are shown in two ways. The first way, shown in row `time`, is the time taken in micro-seconds to perform a unit of redeployment. The second way, shown in row `reductions`, is the corresponding number of reductions that can be performed by the system for a given redeployment. The average number of reductions performed per second by Andorra-I at one processor averaged for all benchmarks is 1920.

	M→S	S→S	S→M
time (μs)	2726	2985	1929
reductions	4.0	4.5	2.9

Table 6.1: AVERAGE RECONFIGURING COSTS

Table 6.2 shows the averaged estimated costs for the activities of and-scheduling and or-scheduling. Column `And` shows the average unitary cost to find and-work in Andorra-I. Column `Or` shows the average unitary cost to find or-work in the system. The two rows show different ways of obtaining the unitary cost of each activity, in micro-seconds and in number of reductions, as in table 6.1.

Table 6.1 shows that it is around one and a half times more costly for a slave to be redeployed to another team than to be redeployed as a master. Table 6.2 shows that the cost to find and-work is far lower than the cost to find or-work. This is explained by the fact that the Bristol or-scheduler is quite complex to perform its functions. Therefore, although it takes fewer reductions to redeploy a worker to be a master, the redeployment is only justifiable if the or-work available contains at least around 10 reductions (cost of redeployment plus cost of

	And	Or
time (μs)	300	5125
reductions	0.5	7.7

Table 6.2: AVERAGE COSTS AND AND OR

or-parallelism) to be performed. On the other hand, if the frequency of and-scheduling is much higher than the frequency of or-scheduling, it might well be better to redeploy slaves to be masters. Although the cost of finding and-work was shown to be quite low, still if there are several slaves in a team, it may be better to start or-work instead of redeploying a worker to help another team, because of the overhead in competing for the run queues of goals. In general, or-work is coarser grained than and-work in Andorra-I, and in a team with several slaves we have more competition for the work existing in the run queues. Therefore, these tables only give a first guideline to start setting values to guide scheduling decisions. When designing any reconfiguring strategy we should take all of it into account.

6.7 Summary

In this chapter we presented the main issues in distributing workers to and-work and to or-work in the context of the Andorra-I system. This chapter had three main objectives: to clearly present the problem to be solved, to present briefly the solution to solve the problem, and given the nature of the solution to the problem, to identify the main sources of overheads of the system and costs to redeploy a worker to and-work and to or-work.

We started by presenting the main problems and limitations encountered when using a fixed configuration of workers into teams in Andorra-I. We claimed that, in order to make Andorra-I a practical system, we needed to *automate* the distribution of work among the workers. In addition, in order to extract maximum parallelism from programs that contain both and- and or-parallelism we needed to use *dynamic reconfiguring* of workers into teams. However, in order to do dynamic reconfiguring, we cannot incur very high overheads, otherwise there will be no benefit in using dynamic reconfiguring. Therefore, we need to design simple and efficient algorithms for the reconfigurer. Moreover, the heuristics for these algorithms should aim at keeping *reconfiguring costs* as low as possible, where reconfiguring costs are a function of other costs in the system. Therefore, in order to distribute processors among and-work and or-work, we need first to identify what are the costs of the system. Based on these costs we can estimate what are the costs to redeploy a worker to an or-task or to an and-task.

The reconfiguring, and-costs and or-costs of the system were tabled after a performance analysis of the system.

We concluded that it is one an a half times more costly for a worker to be redeployed from one team to another than to become a master, because the overheads of redeploying workers as slaves are higher than the overheads of redeploying workers as masters. This means that in order to keep the costs as low as possible, a slave should, whenever possible, be redeployed as a master.

Our performance analysis gave us enough insight on the Andorra-I sources of overheads, and it allowed us to tune the parameters to be used by the reconfiguring strategies.

In the subsequent chapters we describe the strategies and show comparative results of two dynamic strategies for reconfiguring workers among teams. One strategy makes decisions based on estimates of sizes of work obtained at runtime. The other strategy makes decisions based on observation of efficiency of workers, where efficiency is measured as the percentage of time a worker spends working.

Chapter 7

The Work-Guided Strategy

In the previous chapter we discussed the main issues and problems in distributing and-work and or-work in parallel logic programming systems. We showed that the Andorra-I original design was limited and cumbersome to the user. We discussed alternative approaches to distribute and-work and or-work among processors, in order to make the system more friendly and more practical.

In this chapter we discuss the *work-guided* reconfiguring strategy. This strategy has as its main objectives to redeploy workers between and-work and or-work depending on the sizes of the work available at runtime.

7.1 Measures of Work in the Andorra-I tree

In Andorra-I, as described in section 5.2, a worker can be a master or a slave of any team. In a dynamic scheduling scheme we allow free migration of workers from one team to another. A master can be redeployed to another team and become a slave when it runs out of or-work. A slave can be redeployed from its team to another whenever no and-work is found inside its team. A slave can also become a master of a new team.

The main objective of 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- 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, or the number of runnable determinate goals at runtime. As, during execution, at each level of an and-graph, we can have varying numbers of parallel goals to be executed, we can take the width of a whole determinate phase as being the *maximum* width of the whole and-graph, as an upper bound measure. We can also take the width of the determinate phase as being the *minimum* width of the and-graph, as a lower bound measure. Or we can take the width of the determinate phase as being the *average* of all widths happening in the and-graph. Ideally, the number of processors to be allocated to a parallel work of width w must be w. If we consider we could be able to allocate workers for a whole and-graph, the number of workers allocated to the phase could be the average of all widths happening at each level in an Andorra-tree and-graph. However, one could wish to extract maximum parallelism, in which case we would allocate to a determinate phase the number of processors corresponding to the maximum width of the and-graph.

The third measure is the granularity of the determinate phase 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 determinate phase. Goals in a determinate phase have different parallel depths, defined as D_g, where g is a parallel goal. The parallel depth of a 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 determinate phase of the Andorra-tree, in order that each goal can be executed by one processor. Assuming that we have *n* determinate goals, $g_1, g_2, ..., g_n$, that can be executed in parallel, we can estimate the parallel depth of the determinate phase as being $max(D_{g_1}, D_{g_2}, ..., D_{g_n})$, if we want to consider the maximum distance between two choicepoints, an upper bound for the parallel depth of a determinate phase. The parallel depth of a determinate phase can also be considered as being $min(D_{g_1}, D_{g_2}, ..., D_{g_n})$, if we want to estimate the parallel depth as being a lower bound. We can also choose to have a parallel depth of a determinate phase as $avg(D_{g_1}, D_{g_2}, ..., D_{g_n})$, where we would have a compromise between the minimum and maximum granularity of the determinate phase. Notice that this measure varies according to the number of workers that is allocated to each determinate phase.

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 as part of the or-work if this would lead to a faster execution.

Figure 7.1 shows an Andorra-tree containing all kinds of measures of work mentioned. In this tree, each determinate phase contains several and-parallel goals (black dots). In the tree, w is taken as the maximum width of each determinate phase. The parallel depth D is associated with each determinate phase in the Andorra-tree, and corresponds to the parallel depth of the goals that launched the determinate execution, i.e., the root of the subtree corresponding to the determinate phase, so in that case we take the parallel depth of the determinate phase to be the distance between two choicepoints. The amount of or-parallel work *'o'* corresponds to the total number of alternatives for each sub-tree. Note that, ideally, the measures for each subtree could be propagated to its ancestors, and in this case we would have a general picture of the amount of work in the Andorra-tree by only looking at its root node.

When dealing with and-parallel goals, we can have three different situations, where R is the number of processors in the system:

- $R = w$, ideally this is the perfect combination to exploit and-parallelism, if we know that all goals $g_1, g_2, ..., g_w$ can be executed in parallel with each other, and each worker can start executing a goal without any overhead.
- $R > w$, in this case we have workers surplus. $R - w$ processors should be allocated elsewhere.
- $R < w$, this means that $\frac{w}{R}$ units of work that could be executed in parallel need to be executed sequentially, because we do not have enough processors to exploit perfect and-parallelism, ideally.

When we fall in the case where we do not have enough processors to be allocated to all goals (third case above), we need to recompute the new parallel depth of the determinate phase, because, if we are not having perfect parallelism, the ideal parallel depth will

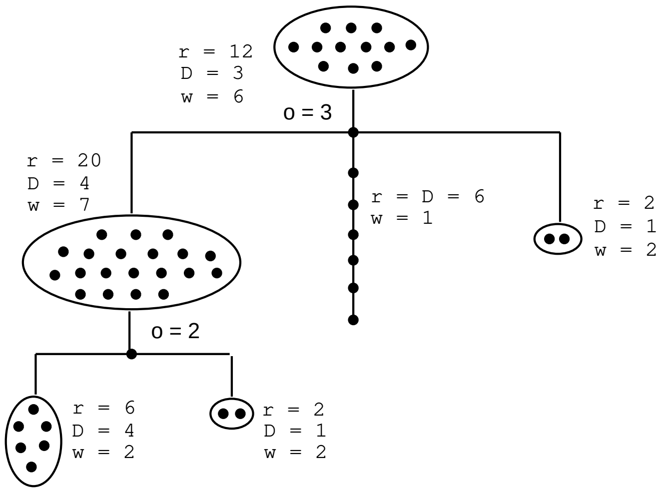


Figure 7.1: ANDORRA-TREE WITH DIFFERENT KINDS OF MEASURES FOR WORK

change, that is the granularity changes. In order to recompute the new parallel depth, we can do:

- compute $a = avg(D_{g_1}, D_{g_2}, ..., D_{g_w})$ (assuming an average bound) of parallel depths for all parallel goals g_i.
- take the ratio $t = \frac{w}{R}$. This is the number of parallel work units that will be executed by each processor, assuming that all w goals can be executed in and-parallel.
- compute the new parallel depth, that is, $t \times a$. This is the granularity size of the work to be done by each processor.

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 discussion we had so far is based on an ideal analysis where we know the 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 a bit 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 as the execution tree is built.

In Andorra-I, the parallel depth of each goal in the run queue and of each alternative in a choicepoint is given as a parameter. For example, if we estimate the parallel depth of each goal to be 1, and we consider that all goals in the run queues can be potentially executed in parallel, the original average parallel depth is given by a, that is equals 1. Then t is the new parallel depth, as $t \times a$ evaluates to t. This value corresponds to the number of goals that will be executed by each worker in a team. For the same reason we do not have an estimate for the parallel depth, we do not have an estimate for the width of determinate phases in Andorra-I. This measure is taken at runtime as being the size of queues of goals as an approximation for the width. 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.

7.2 Description of the Work-Guided Strategy

The main goals of the work-guided reconfigurer are:

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. The preferred team is the one with more and-work per worker, i.e. the one that has greatest parallel depth (better redefined here as the *load* of the team).

As regards (a), in order to find the preferred 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 and-work. This is our Andorra-I's estimate for 'w'. The reconfigurer also takes, from the or-scheduler, the total number of live nodes (nodes that still have alternatives to be taken) or the total number of 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 table 6.1, presented in chapter 6.

In order to estimate the parallel depth of goals and alternatives, we use parameters.

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- with or-work we use the correction factor G_a. In order to compare or- with and-work, we use the correction factor G_o.

The main algorithm for the reconfigurer is presented next. If the worker requesting work is a slave, then:

- If there is neither and-work nor or-work in the tree, the reconfigurer does not take any action, or may release the worker to the system, subject to general conditions such as history of the worker during execution in terms of observation of its activity.
- If there is no and-work in the tree, but there is any amount of or-work, the reconfigurer changes the slave into master. The cost of redeploying a worker to or-work, in this case, is not important, because if there is only one kind of work available, it is not wise to leave the worker idle, so we choose to redeploy the worker to the only kind of work available.
- If there is no or-work, but there is any amount of and-work, the reconfigurer chooses the *preferred team* for the slave, where the preferred team is the one that has more goals per total number of workers, when we add a new slave to the team, i.e., the preferred team is the one that has greatest parallel depth. In this case, the cost to redeploy a worker to and-work is not considered for the same reason as above.
- If both kinds of work are available, and and-work is above the and-threshold, but or-work is below the or-threshold, the reconfigurer chooses the preferred team for the slave.
- if both kinds of work are available, and or-work is above the or-threshold, but and-work is below the and-threshold, the reconfigurer changes the slave into a master.
- If both kinds of work are available, and both and-work is above the and-threshold, and or-work is above the or-threshold, the reconfigurer takes the ratio (t) between the or- and and-work, and compares with the and/or threshold parameter (G_a). If *t* is greater or equal than G_a, the reconfigurer changes the slave into master. Otherwise, the reconfigurer does not take any action.

If the worker requesting work is a master, then:

- If there is neither and-work nor or-work in the tree, the reconfigurer does not take any action, or may release the worker to the system, subject to general conditions of the workers with time.
- If there is no or-work in the tree, but there is any amount of and-work, the reconfigurer chooses the preferred team for the master, and changes the master into a slave. Again, the cost of redeploying a worker to and-work is not considered.
- If there is no and-work, but there is any amount of or-work, the reconfigurer does not take any action.
- If both kinds of work are available, and or-work is above the or-threshold, but and-work is below the and-threshold, the reconfigurer does not take any action.
- If both kinds of work are available, and or-work is below the or-threshold, but and-work is above the and-threshold, the reconfigurer chooses the preferred team for the master, and changes the master into a slave.
- If both kinds of work are available, and both the and-work is above the and-threshold, and the or-work is above the or-threshold, the reconfigurer takes the ratio (t) between the and- and or-work and compares it with the or/and threshold parameter (G_o). If *t* is greater or equal than G_o, the reconfigurer chooses the preferred team for the master, and changes the master into a slave.

The main algorithms that implement the work-guided strategy are shown in Figures 7.1, 7.2, 7.3, 7.4, and 7.5, in a pseudo-code form. The following parameters/variables are used in the algorithms:

- t^*: best team;
- C_a (*and cost*): cost to start and-work;
- C_o (*or cost*): cost to start or-work;
- l (*load*): ratio between the amount of parallelism inside a team (w_t) and the number of workers (R);
- o (*or richness*): amount of or-work;
- w (*and richness*): amount of and-work for all teams;
- d (*delays*): controls the frequency of reconfiguration;
- M (*max delay*): maximum number of delays;

Begin
  Increment d;
  if d $<$ M **then**
    return;
  d := 0;
  Find best team t^**;*
  if there is no such team **then**
    if *decide change to master* **then**
      create a new team;
      return;
    else
      return;
  else
    redeploy slave to team t^*;
End.

Algorithm 7.1: SLAVE CALLS RECONFIGURER

- G_a (*and/or threshold*): parameter to relate the size of and-work with the size of or-work;
- G_o (*or/and threshold*): parameter to relate the size of or-work with the size of and-work;
- D_a (*parallel depth of a goal*): distance between a determinate goal and the next choice-point;
- D_o (*size of an alternative*): number of reductions an alternative may generate;
- R (*number of active workers*): number of current active workers in the system.

As was mentioned before in chapter 6, the reconfigurer is called from the or-scheduler when no or-work is found for a master worker. In this case, the reconfigurer has two alternatives: (a) let the master worker continue to look for or-work; or (b) make the master a slave and redeploy it to the *best* current team. The reconfigurer 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) redeploy this slave from its current team to another better team (the *best* currently available); (c) make the slave a master of a new team.

```
Begin
    Increment d;
    if d < M then
      return;
    d := 0;
    if I am a lonely master then
      if decide change to slave then
        Find best team t*;
        if there is no such team then
              return;
        else
              destroy a team;
              redeploy worker to best team t*;
              return;
      else
           return;
    else
         return;
End.
```

Algorithm 7.2: MASTER CALLS RECONFIGURER

In order to avoid too eager reconfiguration of workers, we maintain a counter that controls when a change is to be made. This is shown by the test `d < M` in algorithm 7.1, algorithm 7.2, and algorithm 7.6. 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 is redeployed to and-work or to or-work.

The algorithms that decide the redeployments for a slave or for a master (algorithm 7.3 and algorithm 7.4) look at two main measures: the amount of parallel work on the tree given by the estimates 'w' and 'o', and the parallel depths of goals (D_a) and alternatives (D_o). The latter estimate gives the granularity of the work to be done by each processor.

Begin
Get 'w' from and-scheduler;
Get 'o' from or-scheduler;
if $(w = 0) \wedge (o = 0)$ **then**
return no change;
if $(w = 0) \wedge (o \neq 0)$ **then**
return change to master;
if $(w \neq 0) \wedge (o = 0)$ **then**
return no change;
if $(\frac{w}{R} \times D_a < C_a)$ **then** /* $(o \geq C_o) \vee (o < C_o)$ */
return change to master;
if $(\frac{o}{R} \times D_o \geq C_o) \wedge (\frac{w}{R} \times D_a \geq C_a)$ **then**
if $(o \geq G_o \times w)$ **then**
return change to master;
return no change;
End.

Algorithm 7.3: DECIDE CHANGE TO MASTER

The amount of or-work (o) is provided by the or-scheduler. It is called the *richness* of a master. All results in this thesis were produced by using the Bristol flexible or-scheduler strategy as discussed in section 5.3.2. Note that 'o' is only an estimate for the size of or-work. We studied the work-guided strategy for two values of 'o'. One is the number of live nodes in the execution tree and the other one is the number of alternatives left open in the execution tree.

The amount of and-work (w) is provided by the and-scheduler and corresponds to the total number of goals in the run queues of all current teams. Note that the size of the run queues is only an estimate of the amount of and-work to be performed in a team. Even if the run queues are empty, goals might be hidden in the suspension goal list. Even if the run queue is not empty, it might only contain non-determinate or sequential goals. Thus a run queue with few goals can be the source of a big production of and-parallel work (determinate goals), while a run queue that has plenty of goals may not be a source of and-parallel work.

Begin
  Get 'w' from and-scheduler;
  Get 'o' from or-scheduler;
  if $(w = 0) \wedge (o = 0)$ **then**
    return no change;
  if $(w = 0) \wedge (o \neq 0)$ **then**
    return no change;
  if $(w \neq 0) \wedge (o = 0)$ **then**
    return change to slave;
  if $(\frac{o}{R} \times D_o < C_o)$ **then** /* $(w \geq C_a) \vee (w < C_a)$ */
    return change to slave;
  if $(\frac{o}{R} \times D_o \geq C_o) \wedge (\frac{w}{R} \times D_a \geq C_a)$ **then**
    if $(w \geq G_a \times o)$ **then**
      return change to slave;
  return no change;
End.

Algorithm 7.4: DECIDE CHANGE TO SLAVE

Workers scan the run queues to find available and-work as discussed in section 5.3.1. 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`. The *and/or threshold* factor (G_a) and the *or/and threshold* factor (G_o), as mentioned before, are correction factors between and- and or-work, and vice-versa.

Algorithm 7.1 describes the procedure that tries to find the best reconfiguration for a slave. The function *create a new team* is just a housekeeping operation that updates the worker and team data structures. The function *Find best team* is shown in algorithm 7.5. The function *decide change to master* is shown in algorithm 7.3.

Algorithm 7.2 describes the procedure that tries to find the best reconfiguration for a master. The function *destroy a team* is just a housekeeping operation that updates the worker and team data structures. The function *decide change to slave* is shown in algorithm 7.4.

Begin
  For each active team t
    Get w_t from and-scheduler;
    $l_t = \frac{w_t}{R_t} \times D_t$;
  end;
  Get team t^* with maxof(l);
  return t^*;
End.

Algorithm 7.5: FIND BEST TEAM

Algorithm 7.3 shows the decision procedure to change the status of a slave to master. It returns a *change to master* value if the slave is to become a master, and a `no_change` value otherwise. A description of this algorithm was given in the beginning of this section.

Algorithm 7.4 shows the decision procedure to change the status of a master to slave. Likewise the previous algorithm, it returns a *change to slave* value if the master is to become a slave. A description of this algorithm was given in the beginning of this section.

Finally, algorithm 7.5 describes the procedure that finds the best team for a worker to join in. All teams are searched for their load. The team with maximum load is chosen.

A variation of the work-guided strategy is to give priority to or-work, by checking the amount of or-work first. If there is or-work available, then ignore the amount of and-work. It can also give priority to and-work, by first checking whether there is and-work available in any other team. If there is and-work then ignore the amount of or-work. We can also have a global decision algorithm that gives no priority to either kind of work. The algorithm is shown in figure 7.6. This algorithm incurs a greater overhead because in all cases we have to get the amount of and-work and or-work and do an exhaustive testing for all possible combinations of amounts of and- and or-work and costs in the same algorithm. This exhaustive testing is done by the function *scheduler_decision*.

If a worker is redeployed to or-work, the best or-work will be found by the or-scheduler according to the or-scheduler strategy used. If the worker is redeployed to and-work, the and-scheduler will find and-work to the worker.

Begin
Increment d;
if d $<$ M **then**
return;
d $:=$ 0;
Get 'w' from and-scheduler;
Get 'o' from or-scheduler;
$decision := scheduler\_decision()$;
if decision = change to master **then**
create a new team;
else
if (*decision* = change teams) or (*decision* = change to slave) **then**
Find best team t^*;
redeploy slave to team t^*;
End.

Algorithm 7.6: GIVE THE SAME PRIORITY TO BOTH KINDS OF WORK

7.3 Implementation Issues

Earlier schedulers were designed based on the software engineering principles of having a clear and distinct interface between engine and scheduler. The reconfigurer was also designed via its interface functions. As a first step, the interactions between the reconfigurer, and- and or-schedulers, and the engine were identified and formed the basis of the interface functions. More details about the basic reconfigurer interface and its implementation can be found in appendix A.

The reconfigurer keeps the sequence of events that happen during execution for future analysis. E.g., whenever the reconfigurer is called, a record of the old and new situations of the system is built containing information about number of teams, what kind of work the worker was given, and all the necessary information to provide compare results from several runs and identify at which points the reconfigurer should be modified and for which applications the behaviour of the reconfigurer is better.

The main data structure used by the reconfigurer is an array of workers. Each entry shown in Figure 7.2 includes information about: flag indicating if it is idle or busy (I/B), type (master - M or slave - S), team which it belongs to (Team) and its history, two fields: one corresponding to the previous type of the worker (hist1), and the other to the number of requests that a worker can make before changing its type (hist2).

Another data structure is used to keep information about the teams. This is a matrix $n \times (n+1)$ whose entries correspond to the number of the teams with the number of workers in that team. The first position of each row contains the size of the team. Every position $n+1$ of each row contains the identifier of worker n.

All other interesting information is passed from engine to reconfigurer and vice-versa, by using the interface macros.

Status	Type	Team	History	
I/B	M/S	Team	hist1	hist2

Figure 7.2: LAYOUT OF A WORKER STRUCTURE.

The data structures are very simple and the algorithms do not incur too much overhead.

7.4 Discussion

Due to some restrictions of the Andorra-I design, the work-guided strategy presented in this chapter is limited by the way workers are redeployed between teams. First of all slaves are just allowed to leave their team if a determinate phase is completed. This simplifies issues related to synchronisation between workers in different teams and between slaves and their master. Another restriction is that workers can only join a team in "stable" moments. This means that if the team is in a transition status (backtracking, or accepting a new member, or preparing for a new execution phase), the worker is not allowed to join the team. This last restriction is also a kind of protection not to let workers join "ghost" teams.

These restrictions in the work-guided strategy are there only to simplify the implementation. Another way of doing things would incur many changes in the main engine and and-scheduler algorithms.

As was mentioned in chapter 5, master workers manage all data structures necessary to execute the Prolog program. Binding arrays belong just to masters, and slaves share the stacks with their masters. Because of this characteristic of the Andorra-I design, every master worker needs to backtrack to the root node when it becomes a slave, because other masters may want to reclaim nodes left behind by the worker that became a master. Thus whenever a master becomes a slave, it needs to clear its binding array in order to be able to use it later in case it becomes a master again. A master that becomes a slave cannot move to the fork node (or pseudo-root, node nearer than the root), because it would block any other worker that would later want to backtrack over that node and reclaim it.

The updating of the binding array is done by moving the master up to the main root of the tree, which means that the worker needs to travel all the way through the nodes of the execution tree, up to the root, and this involves overheads of lockings. Notice that this operation is not constant as it depends on the number of bindings created by the master worker. A better solution for this problem is to do a "lazy" updating. In the worst case the complexity of this operation will be the length of the path from the node where the worker is positioned to the root node of the execution tree, but in the best case it will be just one arc. So, the solution is to keep the slave as a "pseudo-master". Every so often, the slave checks whether its "pseudo-master" position on the tree has been reclaimed. If so, it updates the binding array accordingly. The movement of a master that became a slave around the tree can be a large source of overhead, as the number of nodes to be visited is proportional to the height of the tree up to the current choicepoint. The approach of maintaining the slave as a "pseudo-master" is a sort of lazy binding array updating, that in the average case would help reducing the overheads of moving up and down the tree. No experiment has been done so far to compare the overall overhead of fully moving the worker up to the root with the overall overhead of maintaining the slave as a "pseudo-master".

This solution is quite simple and does not involve a lot of overhead (except the check by the new slaves. It also can prevent slaves from doing some and-work). Another thing is that the new master does not need to reinstall the whole BA anymore once it tries to find or-work again.

In the next chapter, we present the main results produced by our work-guided strategy for a wide range of computations.

7.5 Summary

This chapter presented the *work-guided* strategy to distribute and- and or-work in Andorra-I. This strategy relies on *sizes of work* available in the execution tree in order to distribute and-work and or-work. It also uses information collected from the cost analysis to estimate values for parameters. We described its main algorithms and explained which parameters and estimates of sizes of and-work and or-work are used.

The main parameters used by this strategy are the cost to redeploy a worker to or-work and cost to redeploy a worker to and-work, the relation between and-parallelism and or-parallelism, the estimated *size* of an and-goal, and the estimated *size* of an alternative or of a choicepoint.

The work-guided strategy uses these parameters and the values of size of run queue and number of alternatives or choicepoints in order to decide when to redeploy a worker either to and-work or to or-work.

Lastly we discussed some of the problems and limitations that such strategy might have, and presented possible solutions for them.

Chapter 8

Evaluating the benefit of Reconfiguring and the Work-Guided Strategy

In this chapter we evaluate Andorra-I with the reconfigurer and, in particular, the work-guided strategy. The main objectives of this chapter are to answer the following questions:

- Does Andorra-I with the reconfigurer perform better than Andorra-I as it existed previously?
- Does the reconfigurer perform as well as it could?

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.

The methods for evaluating the work-guided strategy are discussed later in this chapter.

As described in the previous chapter, the work-guided strategy bases its decisions on a set of parameters: cost to redeploy a worker to and-work, cost to redeploy a worker to or-work, different priorities to give work, correction factor between and-work and or-work, size of goals and size of alternatives. We made experiments for a wide range of combined parameter values. In this chapter we present results obtained with a single parameter setting that produced the best results over all the benchmark set. Initial values for some of the parameters were taken from the values shown in table 6.1, chapter 6. A range of integer values greater than and lesser than those values were tried. Not all combinations of parameter values were tried as this would lead to an infinite set of experiments. Our main conclusion for the parameter settings tried is that although, for some benchmarks, some variation of parameter values can affect performance, we could choose overall one single set of parameters that produced good performance on all benchmarks. Some experiments with different parameter settings are shown in appendix B. More detailed study of variation of parameters for ten processors is shown in appendix C.

Another important parameter is the starting configuration to run Andorra-I with the reconfigurer. The starting configuration could have been obtained through compile-time analysis of the programs. But sometimes compile-time analysis would not be able to infer any starting configuration due to the complexity of the problem. In that case we would be back to the same problem of having to choose different configurations for each computation. The reconfigurer, therefore, uses a standard initial configuration, which is chosen to be a single team (i.e., a configuration that supports and-parallelism, but not or-parallelism). This was an arbitrary choice, but as we will see later, this choice for the starting configuration does not cause any impact on the performance, because the reconfiguring overheads are very low.

In the first section of this chapter, we present the benchmark set used in this thesis. In the next section we discuss the benefits of using a reconfigurer in Andorra-I. In the third section we evaluate the work-guided strategy, and in the last section, we present a summary of the chapter.

8.1 About the Benchmarks

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. A list of the benchmarks can be found in appendix D.

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 following description, benchmarks are presented in the following order: (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. Later these groups of programs will be broken into subgroups in order to evaluate the performance of the work-guided strategy more closely.

8.1.1 Artificial Benchmarks

ndetdet

This program is written specially to test the reconfigurer. It starts with a non-determinate phase which is followed by a determinate phase. The ideal Andorra-I tree for this program can be seen in figure 8.1. There are 10 alternatives for `q/2` (which expand to the bodies `s1/2, s2/2`, etc), and 4 alternatives for each `s/2`. From these 4, three fail immediately and the leftmost succeeds producing some amount of and-work through `l/1` predicates (this predicate simply simulates loops of size N for `l(N)`). This program presents a large degree of or-parallelism and a reasonable amount of and-parallelism.

mixed

This is another program written to test the reconfigurer. It mixes both and- and or-parallelism in an irregular way. The and-parallelism in this program is very fine grained. Or-parallelism is predominant.

detndet

This is yet another program written with the special purpose of testing the behaviour of the reconfigurer and Andorra-I. It starts with a determinate phase followed by a non-determinate phase. The program is shown in program 8.1, and the respective tree for the query `p(X,Y)` is shown in figure 8.2. This program contains mainly or-parallelism with a small amount of and-parallelism.

The goal `s(a,b)` starts the first determinate phase by solving all `loop/1` predicates. When they are all solved a choicepoint is created for `r/2`. This choicepoint creates 90 alternative clauses in the node, each one with a computation size given by `loop(100)`.

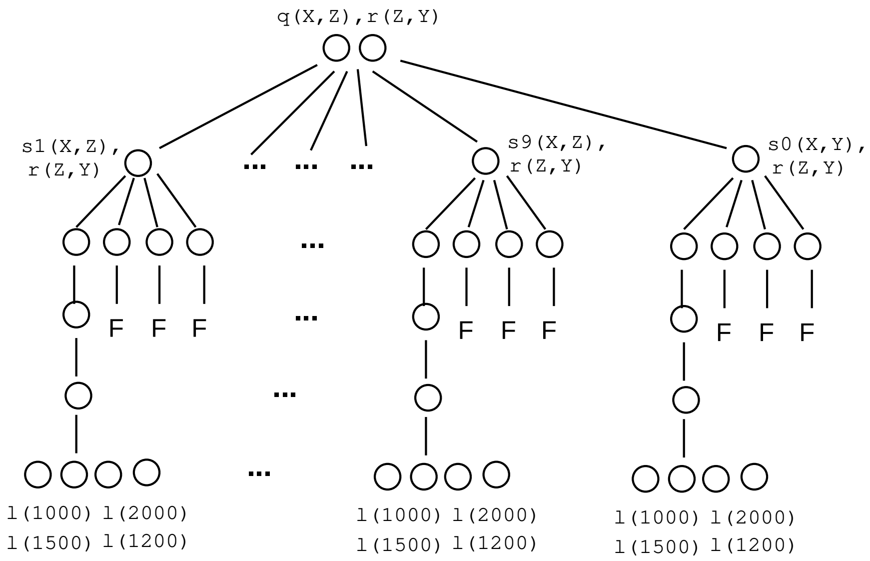


Figure 8.1: ANDORRA-I TREE FOR PROGRAM `ndetdet`

8.1.2 Benchmarks with predominantly or-parallelism

bqu6 and **bqu8**

This is a program to solve the Queens problem written using a Pandora programming technique [4]. The problem is to place queens in a board (NxN) in order that no queen attacks each other in any column, row or diagonal. Or-parallelism arises when trying to solve the `cell` predicate. The boards sizes tested were 6x6 and 8x8. The program has mainly or-parallelism with a small amount of and-parallelism.

chat

This is an example from the well-known natural language question-answering system Chat-80, written at the University of Edinburgh by Pereira and Warren [141]. This version of Chat-80 operates on the domain of world geography. The program `chat` makes a query to the Chat-80 database. It contains only or-parallelism, and it has been used as one of the or-parallel benchmarks for both the Aurora and Muse systems.

cypher

This is a simple substitution decoding system developed by Rong Yang [146] in our Andorra-I group. The system reads an encrypted text and decodes it (assuming the original message is in English). According to the statistics, if one knows about 100

```
p(X,Y) :- s(X,Z), r(Z,Y).

s(a,b) :- loop(2000), loop(500), loop(1500),
          loop(10), loop(100), loop(50), loop(500),
          loop(2000), loop(30), loop(90).

loop(0).
loop(N) :- N > 0,
           New_N is N - 1,
           loop(New_N).

r(b,c) :- loop(100).   r(a,d) :- loop(100).         ....
r(a,b) :- loop(100).   r(b,e) :- loop(100).   r(b,c) :- loop(100).
r(a,c) :- loop(100).   r(b,f) :- loop(100).   r(a,b) :- loop(100).
```

Program 8.1: `detndet` PROGRAM

of the most common words, one can understand on average 60% of most text. Now, instead of having an entire English dictionary, we can solve a cipher using only a very small dictionary of common words (about 150 words). With this dictionary, the Andorra-I preprocessor can generate a reasonably small determinate tree for each word of a set length from the dictionary. Then the program performs a lot of letter matching, first determinately in and-parallel, then, when only non-determinate matching is left, the program creates several choices (giving rise to or-parallelism). An or-branch fails when the number of unmatchable words exceeds a certain limit. The algorithm takes advantage of the basic Andorra model to greatly reduce the search space.

The length of the ciphertext used in the benchmark is 56 letters, and the maximum number of unmatchable words is set to 2. This program has mainly or-parallelism with a small amount of and-parallelism.

mutest

This is a program that solves a problem described in [73]. The problem is to produce the string ’mu’ from a given string, through production rules that can be applied in any order. The theorem to prove is that ’miiiii’ produces ’mu’. If we replace the letters in the strings by numbers, this puzzle solves some interesting problems in number theory. This program contains only or-parallelism.

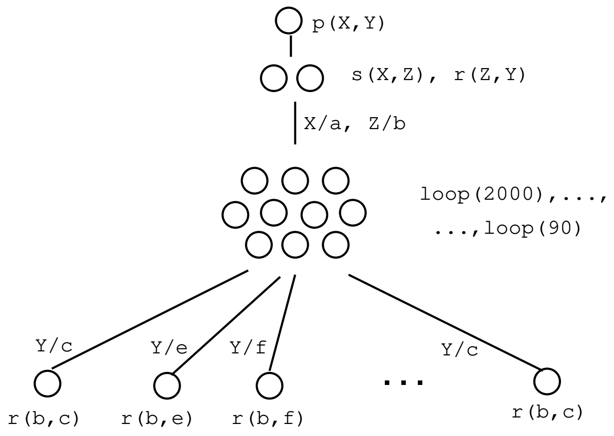


Figure 8.2: ANDORRA-I TREE FOR PROGRAM `detndet`

8.1.3 Benchmarks with predominantly and-parallelism

cross6 and **cross11**

These are two input sets for a program that solves crossword problems according to the constraint technique used by van Hentenryck in [130]. A crossword generator was written with the purpose of automatically producing different kinds of crosswords to test Andorra-I. The crossword generator program accepts as an input the crossword format, horizontal words to be placed in the crossword and vertical words to be placed in the crossword. The crossword programs produced to run in Andorra-I have a crossword of 8x8 (`cross6`) and 12x12 (`cross11`). In the test program generated by the crossword generator, the crossword itself is expressed by a clause with a huge body defining the shape of the crossword in terms of horizontal and vertical blank spaces to be filled in the crossword. Each word is numbered and defined as a database of facts with one fact per letter in the word. The data sizes for this program are too small to produce a reasonable amount of parallelism. We could not run this program with a bigger data size, i.e., bigger crossword dimension, because of memory limitations in Andorra-I. And-parallelism arises from the attempt of putting words on the crossword. Or-parallelism arises when selecting letters of the words to put on the crossword.

flypan2, **flypan4** and **flypan5**

These are three different input sets for a program to generate naval flight allocations, based on a system developed by Software Sciences and the University of Leeds for the Royal Navy. It is an example of a real life resource allocation problem. The program allocates airborne resources (such as aircraft) while taking into account a number of constraints. The problem is solved by using the technique of active constraints as first implemented for Pandora [3]. In this technique, the co-routining inherent in the Andorra model is used to activate constraints as soon as possible. The program has both or-parallelism, arising from the different possible choices, and and-parallelism, arising from the parallel evaluation of different constraints. We used three input data for testing the program. The first one consists of 11 aircraft, 36 crew members and 10 flights needed to be scheduled, the second one consists of 14 aircraft, 49 crew members and 10 flights needed to be scheduled, and the third one consists of 14 aircraft, 70 crew members and 10 flights to be scheduled. The degree of and- and or-parallelism in this program varies according to the queries, but all the queries give rise to more and-parallelism than to or-parallelism.

bt_cluster

This is a clustering algorithm for network management from British Telecom [29]. The program receives a set of points in a three dimensional space and groups these points into *clusters*. Basically, three points belong to the same cluster if the distance between them is smaller than a certain limit. To obtain best performance, we rewrote the original application to become a determinate-only computation. And-parallelism only in this case naturally stems from running the calculations for each point in parallel. The test program uses a cluster of 400 points as input data. This program has no or-parallelism.

scanner

This is a scanner program to reveal the contents of a bitmap. The program is an AKL [76] benchmark developed at SICS. The problem was described in [38]. It reveals the contents of a box (bitmap). The input is the number of dots on each row, column, left diagonal, and right diagonal. The bitmap used in the benchmark is a picture of a star. The program contains both or-parallelism corresponding to different choices for the unknown bits (either a dot or a blank), and and-parallelism corresponding to parallel propagation of determinate bits and evaluation of constraints. This program gives rise to reasonable amounts of both and- and or-parallelism in interleaved phases.

road_markings

This is a program to extract road markings developed by Hluchoweckyj [72] from the vision research group, in the department of Computer Science at the University of Bristol.

This program is part of a Ph.D. project to extract road markings from real world images that have been processed to identify a set of visible objects. The algorithm fits either straight lines or quadratic curves through these objects. The "best" set of curves is taken as being the road boundaries, where "best" is defined by certain heuristics such as the width between two road boundaries being constant and greater than a vehicle-width apart. This application contains and-parallelism corresponding to parallelising the operations on each pair of objects. The testing program takes 16 objects as the input data. This program has mainly and-parallelism arising from the different objects and a very small amount of or-parallelism.

bcnet

This is another real world application developed by British Telecom [95]. It was developed on the scope of the BT contract with the Bristol University. The application involves finding paths through a network from a common source node to several destination nodes in such a way that the overall cost is minimised. The cost of using an individual link may reflect factors such as its length, its capacity, the average level of its demand, or the time of day at which it is demanded. Therefore the shortest route might not always be the cheapest. The version of the program used as a benchmark uses the Andorra-I constraints facilities [53].

8.2 Andorra-I with the Reconfigurer

In this section we intend to show the benefits of adding a reconfigurer to the Andorra-I system. Our objective is to show that Andorra-I with a reconfigurer is much more user friendly, while achieving very good performance. In order to do that we compare 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 as we take real results produced in practice with plausible fixed configurations.

There is an infinite number of fixed configurations possible. We will limit our comparison to five 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.

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

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 five plausible configurations:

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

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

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 can have $\sqrt[2]{n}$ teams with $\sqrt[2]{n}$ workers each.

4) give weight 2 to and-work and weight 1 to or-work. In this case $p = \frac{1}{3}$ and $1 - p = \frac{2}{3}$. In that case we can have $\sqrt[3]{n}$ teams with $\sqrt[3]{n^2}$ workers each.

5) give weight 1 to and-work and weight 2 to or-work. In this case $p = \frac{2}{3}$ and $1 - p = \frac{1}{3}$. In that case we can have $\sqrt[3]{n^2}$ teams with $\sqrt[3]{n}$ 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, two teams with 2 masters with 4 slaves each for weighting 4, and 5 teams with 5 masters with 1 slave each for weighting 5. The size of the teams for different choices gives the desired balance to exploit and-parallelism and or-parallelism.

Table 8.1 shows speedups achieved at 10 processors with the plausible fixed configurations, and the speedups achieved with the reconfigurer. The five middle columns of the table give the speedups of the benchmark set for the five plausible fixed configurations, in increasing order of p. The column `Reconfig` gives the speedups achieved by the reconfigurer. Shaded rectangles 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 five fixed configurations. The reconfigurer reaches an overall result that is around 55% better ($\approx$ 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 any of the benchmarks.

Benchmarks	Speedup with weighting, p =					Speedup with Reconfigurer
	0	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{2}{3}$	1	
ndetdet	2.037	4.673	5.913	7.971	8.487	9.831
mixed	0.980	1.920	2.642	3.819	3.584	4.004
detndet	1.044	1.622	2.052	2.387	2.032	3.157
bqu8	1.527	2.858	3.937	5.487	7.516	4.474
chat	0.666	1.578	2.502	3.866	7.334	7.034
cypher	1.147	2.164	2.875	4.046	5.922	6.116
mutest	0.584	1.386	2.071	3.450	5.959	6.025
bqu6	1.499	2.562	2.940	3.132	3.137	2.920
flypan2	6.203	4.105	3.507	1.964	1.007	6.307
bt_cluster	6.002	4.353	3.947	1.995	1.011	5.793
flypan4	5.079	3.725	3.305	1.947	1.062	4.961
scanner	5.009	5.460	4.992	4.087	3.226	5.414
road_markings	4.887	3.452	2.479	1.790	0.965	4.269
bcnet	3.226	3.763	3.336	2.767	2.033	3.942
cross11	2.666	1.647	1.615	1.669	1.112	2.917
flypan5	2.629	2.878	2.506	2.139	1.335	3.382
cross6	2.529	2.260	2.095	1.342	0.964	2.260
H.mean	1.656	2.505	2.787	2.574	1.836	4.258

Table 8.1: ANDORRA-I IN THE PAST X ANDORRA-I WITH THE RECONFIGURER, AT 10 WORKERS

From the figures shown in table 8.1, 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.
- 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 `flypan5`.
- 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`, `flypan4` and `scanner`, the performance obtained with the reconfigurer is comparable 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.

8.3 Performance Evaluation of the Work-Guided Strategy

After showing that Andorra-I with the reconfigurer performs much better than Andorra-I without the reconfigurer, in this section 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 already used in the previous section 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.

The first method consists of devising a mathematical model for the computations, where a mathematical formula would give an estimate of a possible optimal speedup for a computation. This method is not suitable for logic programs, because the computations are very irregular, therefore it would be difficult to find a good mathematical model to describe the computational behaviour of a program.

The second method is a whole subject by itself, and not very trivial. The idea is to simulate the parallel model and estimate optimal speedups for the computations, with limited and unlimited numbers of processors. The simulation can be done at a high level, i.e., only simulate the conceptual model, or it can be done at a low level. Low level simulation is more realistic because it takes into account implementation issues of the system. Work done by Shen [110] provides such a tool, but for studying the exploitation of or-parallelism combined with non-determinate independent and-parallelism, which is a very different model from what we are using. Also Fernandez [47] studies ideal speedups for the same model. Both of them predict performance of a model that combines or-parallelism with independent and-parallelism at unlimited number of processors, and at a limited number of processors by finding a quasi optimal schedule to run the applications. Sehr [106] uses a yet different method to achieve the same goal, by annotating the Prolog program with special predicates and extra argument variables to pass around goal and clause time information.

The third 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 it 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.

	Target/Cfg		Reconfig
ndetdet	8.487	*(10M)*	9.831
mixed	3.997	*(6M4S)*	4.004
detndet	3.012	*(8M2S)*	3.157
bqu8	7.516	*(10M)*	4.474
chat	7.334	*(10M)*	7.034
cypher	5.922	*(10M)*	6.116
mutest	5.959	*(10M)*	6.025
bqu6	3.950	*(9M1S)*	2.920
flypan2	6.203	*(9S)*	6.307
bt_cluster	6.002	*(9S)*	5.793
flypan4	5.079	*(9S)*	4.961
scanner	5.485	*(2M8S)*	5.414
road_markings	4.887	*(9S)*	4.269
bcnet	3.746	*(2M8S)*	3.942
cross11	2.666	*(9S)*	2.917
flypan5	2.862	*(2M8S)*	3.382
cross6	2.529	*(9S)*	2.260
H.mean	4.407		4.258

Table 8.2: BEST FIXED CONFIGURATION X RECONFIGURER, AT 10 WORKERS

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 table 8.2 speedups achieved for the best fixed teams configuration of 10 workers for each of our benchmarks. The second column of the table (`Target/Cfg`) shows the speedups achieved by the best fixed configuration. 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 third column of the table (`Reconfig`) shows the speedups achieved with the work-guided strategy. The results shown for the reconfigurer were produced with a single version of Andorra-I, where the reconfigurer starts from a configuration of only one team of workers. The last row of the table (`H.mean`) shows the harmonic mean over all the benchmarks.

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.

Next we evaluate the work-guided strategy up to 15 processors. In an earlier paper [43] we described briefly the initial work-based strategy and showed some results for a very small number of processors and different benchmarks, for the interpreter-based Andorra-I.

In order to reduce the complexity of this section we present graphs for some representative classes of programs of our benchmark set. They fall into six subgroups depending on whether the parallelism is high or low, and whether it is mainly and-parallelism or or-parallelism. For brevity, we call these groups: (1) *only or*, (2) *only and*, (3) *high or, low and*, (4) *high and, low or*, (5) *low or, lower and*, and (6) *low and, lower or*. Group (1) has only or-parallelism. Group (2) has only and-parallelism. Group (3) has a high degree of or-parallelism with low degree of and-parallelism. Group (4) has a high degree of and-parallelism, and low degree of or-parallelism. Group (5) has a low degree of or-parallelism and low degree of and-parallelism, but the amount of or-parallelism is bigger than the amount of and-parallelism. Group (6) has a low degree of and- and or-parallelism, but the amount of and-parallelism is bigger than the amount of or-parallelism.

The graphs in figures 8.3, 8.4, 8.5, 8.6, 8.7 and 8.8 show the comparison between the work-guided strategy and our target performance. The thick line in the graphs represents the performance of the work-guided strategy, while the thin line represents our target performance. Note that for some of the benchmarks, we need different weights for and- and or-, which means that we may need different configurations of masters and slaves at different numbers of processors in order to achieve best speedups with a fixed configuration. The fixed configuration that produced the best speedup is attached to the fixed configuration curve (these are shown as small numbers in the graph. For example, `11M3S` in figure 8.5 shows that, at 14 processors, the best fixed configuration for the benchmark `ndetdet` is

Figure 8.3: WORK-GUIDED STRATEGY: ONLY OR BENCHMARKS

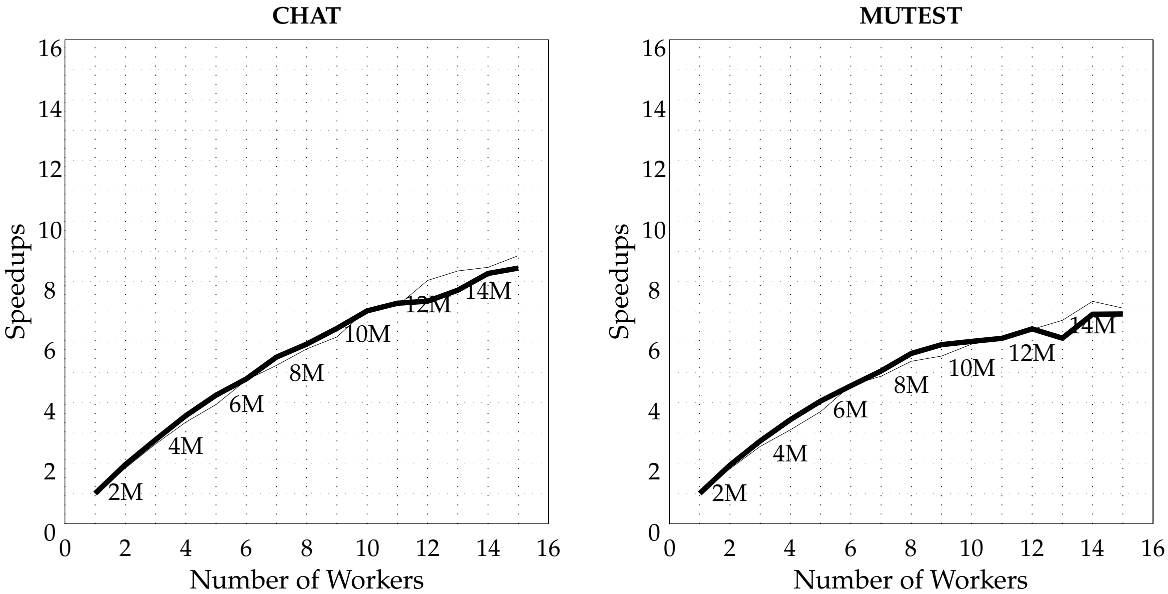
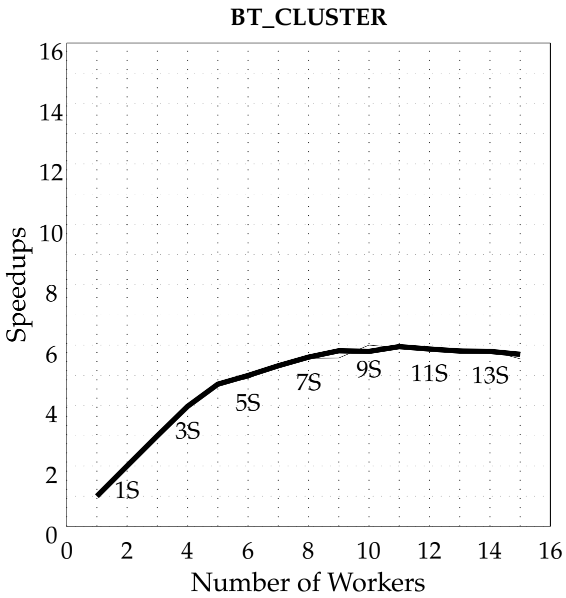


Figure 8.4: WORK-GUIDED STRATEGY: ONLY AND BENCHMARK



the one with 11 teams, three of them with 2 workers and the remaining with only one worker).

Computations that contain or-parallelism only achieves best performance with a fixed configuration tuned to exploit or-parallelism only, i.e., several teams with only one master (choice of weighting 1). Computations that contain and-parallelism only achieve best performance with a fixed configuration tuned to exploit and-parallelism only, i.e., one team of one master and several slaves (choice of weighting 2). The work-guided strategy performs similarly to these different versions of Andorra-I, as is shown by benchmarks `chat` and `mutest` in figure 8.3, and benchmark `bt_cluster` in figure 8.4. This means that the overhead of rearranging workers into teams is very low.

For programs that contain a mixture of both kinds of parallelism, we would like the work-guided strategy to perform similarly to or better than the target performance, although this is a difficult target. Indeed the reconfigurer reaches the target performance for most of the benchmark programs that contain a mixture of both forms of parallelism (e.g., `ndetdet` in figure 8.5, `flypan2` in 8.6, and `bcnet` in figure 8.8).

The computation `flypan2` (figure 8.6) is an example of a program where no fixed configuration would achieve best performance, because it contains two distinct phases of computation, an and-parallel phase followed by an or-parallel phase. In practice, with this particular class of computations (that includes all computations that contain distinct phases), we showed that using dynamic reconfiguration of workers into teams we achieve better speedups than setting the best fixed configuration. This is also true for the benchmark `detndet` (figure 8.7).

For the program `flypan2`, we can show by simple analysis that our strategy obtains the best speedup possible with the Andorra-I implementation. As program `flypan2` has two distinct phases of computation, we can say the program is executed optimally in parallel by Andorra-I within the following time:

$$D_p + N_p$$

where D_p is the time taken to execute the and-parallel phase in parallel, and N_p is the time taken to execute the or-parallel phase in parallel, i.e., the determinate phase taking D_p time units to finish and the non-determinate phase taking N_p time units to finish. Let D_s be the time to execute the and-parallel phase sequentially, and N_s be the time to execute the or-parallel phase sequentially. From our experiments with Andorra-I, taking as an example runs for 10 processors, we have the following time information for this program:

(1) $D_s + N_s = 13.727 secs$, this corresponds to the average time taken to execute the program sequentially;

Figure 8.5: WORK-GUIDED STRATEGY: HIGH OR, LOW AND BENCHMARKS

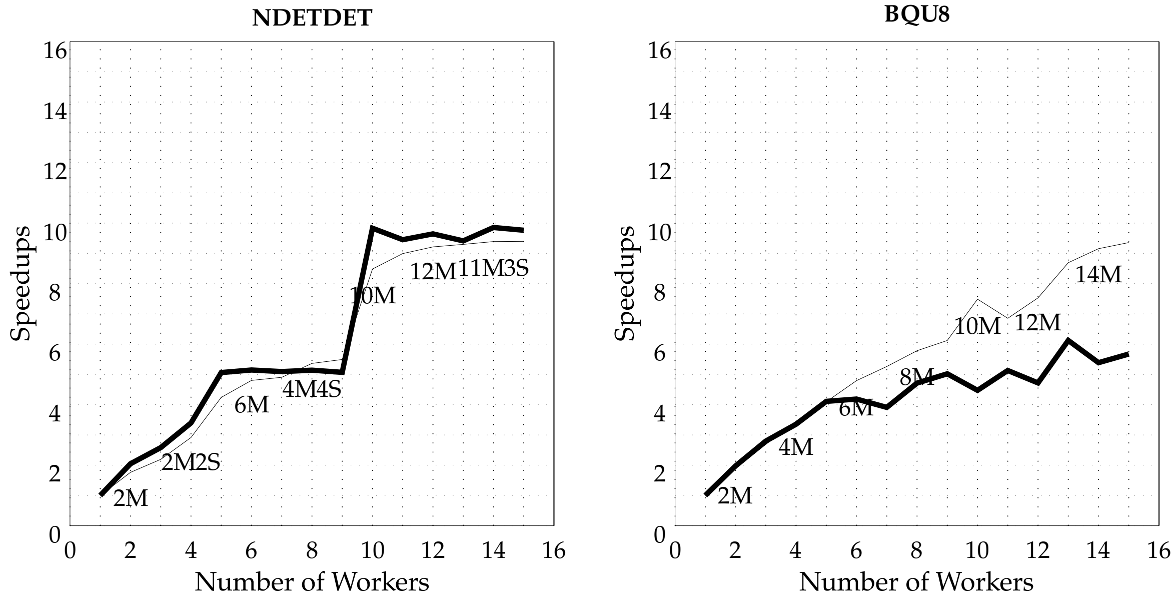


Figure 8.6: WORK-GUIDED STRATEGY: HIGH AND, LOW OR BENCHMARKS

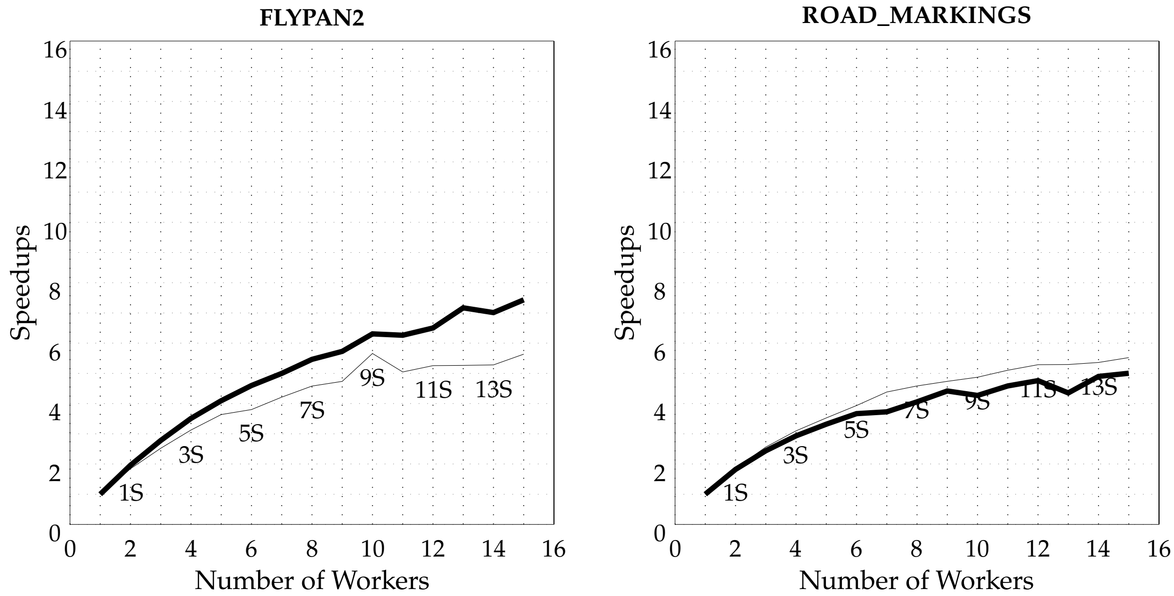


Figure 8.7: WORK-GUIDED STRATEGY: LOW OR, LOWER AND BENCHMARKS

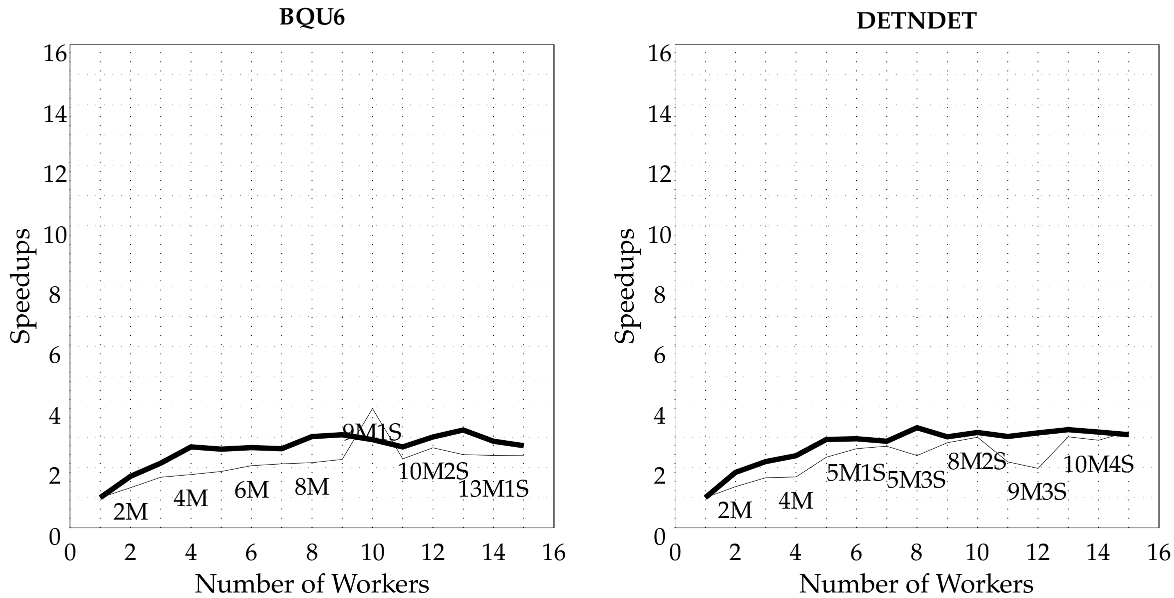
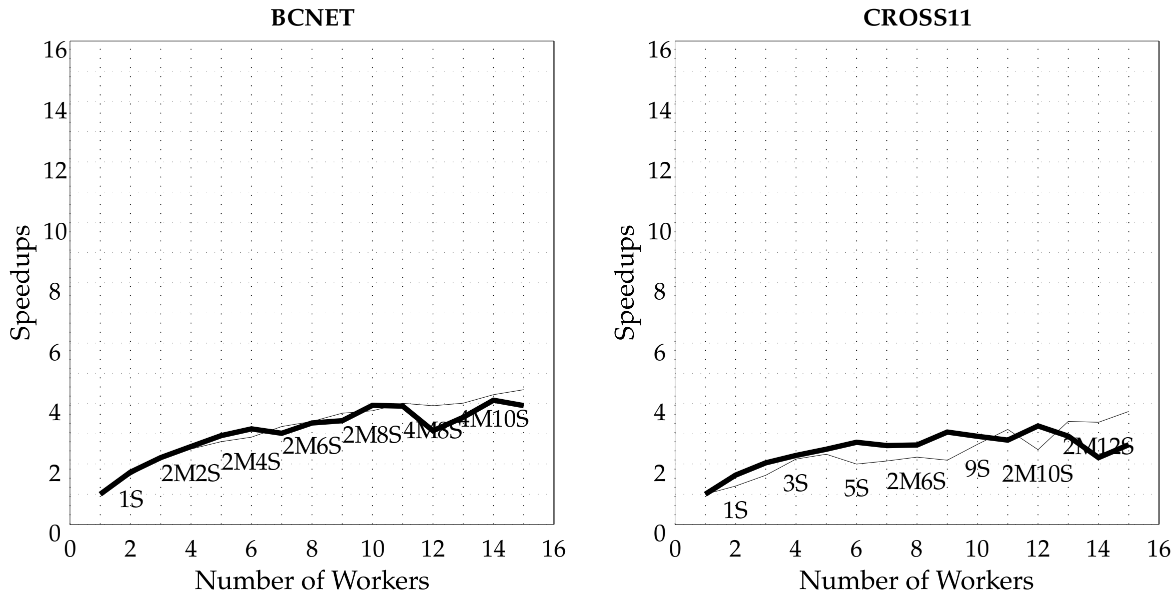


Figure 8.8: WORK-GUIDED STRATEGY: LOW AND, LOWER OR BENCHMARKS



(2) $D_p + N_s = 2.422 secs$, this is the average time taken to execute the program with only and-parallelism, at 10 processors;

(3) $D_s + N_p = 13.646 secs$, this is the average time taken to execute the program with only or-parallelism, at 10 processors.

Adding (2) and (3), we get:

$$D_s + N_s + D_p + N_p = 16.068$$

Subtracting (1), we get,

$$D_p + N_p = 2.341$$

i.e., the time to execute the program optimally with and-parallelism and or-parallelism with 10 processors should be around 2.341 secs. The best fixed configuration of workers in Andorra-I executes this program at 10 processors in time 2.422 secs, while the work-guided strategy produces execution time of 2.172 secs. This means not only that the reconfigurer produces a better speedup than the target speedup, but it manages to achieve slightly better speedups than the best predicted by the preceding analysis. This is explained by the fact that, if there is little and-parallelism available, and there are more workers in a team than necessary to exploit the parallelism, Andorra-I behaves very inefficiently. Idle slaves in a team can steal the work of any other worker, and the execution may be swapped from one worker to another very frequently. The immediate disadvantage of this phenomenon is that processors need to reload their caches frequently. While using fixed configuration of workers, Andorra-I incurs more overheads with workers in a team competing to execute the work available. While using dynamic reconfiguring, instead of competing to obtain more work in its team, a worker has a chance of being redeployed to elsewhere, and do not disturb the execution of other workers. As the reconfigurer prevents slaves from disturbing each other by redeploying them to elsewhere, and the total cost of reconfiguring is much lower than the costs of sharing parallel and-work or parallel or-work in a fixed configuration, the reconfigurer achieves slightly better speedups than the best possible computed from real runtime units.

If we apply the same reasoning to program `detndet`, the result is similar, with the reconfigurer performing around 10% better in average than the best possible, predicted analytically.

For some of the computations (e.g., `bqu8`, in figure 8.5), the work-guided strategy achieves good performance, but it does not reach the target performance. This is explained by the fact that the or-parallel phase in these computations is dominant, but there is a very small amount of and-parallelism in the or-parallel branches. Because the reconfigurer uses instant runtime information to redeploy workers, it believes that there is more and-parallelism than the program really contains (although the run queue has plenty of goals, they are finished very shortly), and prevents slaves from becoming masters to exploit the or-parallelism available.

8.4 Discussion

We demonstrated the benefits of automating the distribution of and-work and or-work in Andorra-I. In addition, as we designed a *dynamic* reconfigurer, we had the extra benefit of obtaining speedups better than the best fixed configuration could achieve for some classes of applications.

Purely determinate programs are easy to handle: as there are no choicepoints being created during execution, all slaves will remain in their original team, because obviously or-richness will be zero. If the determinate application does not present a good amount of and-parallelism to be exploited, slaves will be idle looping till the end of the execution. This would be a good opportunity to release workers to the system.

Purely non-determinate programs with sequential branches that do not lead to and-parallel work are also easy to handle. Even if the system default is to start with a single team of workers, they will automatically reconfigure to create new teams and start new or-work. This happens because the size of the run queues is very small and the slaves become idle almost immediately. Then they have a chance to find or-work or in a future implementation to be released to the system.

It follows from both examples above that if we have a program whose structured execution tree has a determinate phase followed by a non-determinate phase, the workers will rearrange automatically for the second phase.

Non-determinate programs bring a drawback to the work-guided strategy if some of the branches present a small amount of and-parallel work. In the beginning, some idle slaves will become masters and grab alternatives from the first leader worker (at this time, there is no other source of work in the system, except the choicepoint made by the leader worker). Later, when the branches owned by the newly made masters are started, some and-work might arise from their branches. The other slaves that are still idle looping will be redeployed to help to finish the and-work in the branches, however this amount of and-work may not be suitable for being executed in parallel. In that case there will be a slow down to the execution, because workers in a team are competing to execute work that is not and-parallel.

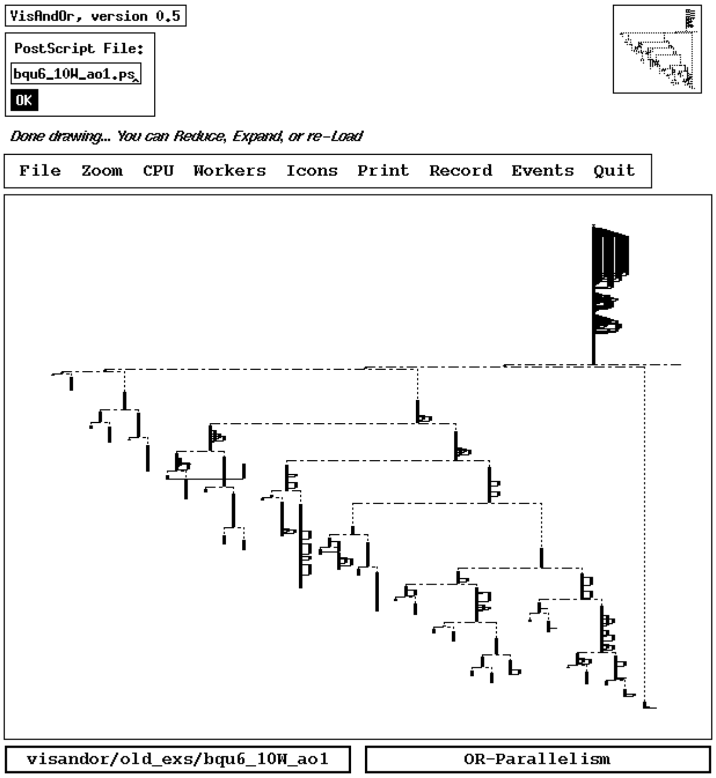


Figure 8.9: ANDORRA-I TREE BY VISANDOR FOR `bqu6` AT 10 WORKERS

Programs that mix both types of parallelism at the same time are the main challenge for the reconfigurer. If the program does not present a reasonable grain size for its tasks, both and- and or-tasks, Andorra-I would not be able to exploit enough parallelism, but if the tasks have a reasonable grain size, the system should be able to exploit the parallelism. It remains for the reconfigurer to guarantee that the resources will be well utilised to exploit all parallelism available. Unfortunately this is not so easy a task. With the work-guided strategy the workers are redeployed frequently between and- and or-work. It is a "short-sighted" strategy.

In summary, 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 in this chapter, the work-guided strategy does not reach the target performance for this particular kind of problem, typified by the benchmark `bqu8`.

As discussed before in this chapter, 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. Figure 8.9 depicts the parallelism in a simple version of the Queens program to solve a board 6x6. The picture was produced with the VisAndOr tool developed in the Polytechnic University of Madrid by Luis Gómez [51]. The graph shows a static trace of the Andorra-I execution for `bqu6` with 10 processors. The time-line grows from the top to the bottom. A fork in a continuous vertical line represents or-parallel work, and in a single vertical line originated from a fork, there can be attached parallel lines that indicate slaves are helping a master. The picture shows a very small determinate phase in the beginning with all slaves helping the execution, and a large amount of choicepoints in the second phase of the computation. Notice that in some branches, there are some slaves trying to help their masters, but they do not stay in a team for very long, because they become idle and are redeployed to somewhere else. The computation `bqu8`, solving the queens problem for a board of size 8x8 has a much bigger non-determinate phase. 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.

We suggest that another heuristic to distribute and- and or-work may work better than the work-guided strategy where the work-guided strategy does not reach the target performance. In the next chapter we present a new strategy that tries to cope with this problem by distributing work among processors according to the degree of utilisation of each processor, instead of estimates of and-work and or-work.

In the beginning of this chapter we discussed that the work-guided strategy uses some parameters in order to distribute workers between and-work and or-work. We presented in this chapter results that were obtained with a single set of parameters that produced overall reasonable performance. We believe that eventually we could find a set of parameters that could produce better speedups than the ones shown, however the improvement would not be very significant. For programs that do not present much parallelism, a new set of parameters would not make much difference. For some of the programs that contain parallelism, another parameter setting would produce some improvement, however this improvement for some of the programs might decrease the performance for other programs.

8.5 Summary

In this chapter 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 the work-guided reconfiguring strategy described in the previous chapter.

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:

- The user does not have the burden of deciding which configuration to use.
- Overall, the reconfigurer performs much better than any one of the fixed configurations.
- For individual benchmarks, the reconfigurer performs as well as or better than any of the fixed configurations.

We also attempted to evaluate whether the performance of the work-guided strategy is as good as it could be by comparing it with a target performance given by the best speedup achievable by any fixed configuration.

The comparison with the target performance depends on the kind of application. For programs that present only one form of parallelism, the work-guided strategy performed similarly to the target performance, which implies that rearrangement of workers into teams does not incur much overhead. For programs that present both kinds of parallelism with a reasonable grain size, the work-guided strategy performs around 6.5% better in average than the target performance, at 10 processors. For programs that contain both kinds of parallelism, but with fine grain size, the work-guided strategy still performs better than the target performance for some cases, is similar to the target performance in other cases, and does not perform very well for certain cases.

As there are some cases where the work-guided strategy does not match the target performance, we suggest that there is still some room for improvements.

Chapter 9

The Efficiency-Guided Strategy

In the previous chapter we studied the benefits of using a reconfigurer in Andorra-I and also evaluated the performance of the work-guided strategy. Our performance evaluation of the work-guided strategy was done by comparing it with the old version of Andorra-I without the reconfigurer. In order to evaluate how well the work-guided strategy performed, we compared it with a *target performance* defined as being the best performance achievable with the best fixed configuration adjusted for each application at each numbers of processors.

Although we proved experimentally that the work-guided strategy performs much better than any plausible fixed configurations, the work-guided strategy does not reach our target performance for a particular benchmark. The reason for that is that, by design, the work-guided strategy makes decisions based on instant information about amounts of and-work and or-work. This means that this strategy is somehow "short-sighted", which, in some cases, leads to too many unnecessary reconfigurations. Particularly if the parallel work is very fine grained or if the work-guided strategy takes the wrong runtime information concerning the sizes of and-work and or-work.

In order to overcome this drawback, we designed another strategy whose aims are to avoid the short-sightness of the work-guided strategy, and to be more conservative in reconfiguring. This new 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, for distributed network systems.

Therefore, in this chapter we present another way of doing dynamic reconfiguration of workers into teams: the *efficiency-guided* strategy.

9.1 Motivation

The work-guided strategy described in chapter 7 relies on amounts of work existing in the teams or in the choicepoints to decide when to redeploy a worker. That strategy proved to produce very good results compared with several plausible fixed configurations choices. The advantage of this strategy is that if the size of the work taken at the moment of the decision to redeploy persists along the time, it is likely to work well. On the other hand, if the work often varies, or if it is very fine grained, or if the estimators of sizes of work are not correct, the work-guided strategy may not work satisfactorily.

One ideal solution to improve the accuracy of the strategy would be to efficiently compute at runtime an accurate estimate of the size of the and- and or-work available or still to be created. As discussed before, in order to obtain this information, the system would need the aid of compile-time granularity analysis. Researchers have been working in this direction [151, 82, 49], and the literature reports some improvements in the area, however no system has in fact used such information to do scheduling of any kind. In practice, grain sizes can be estimated for some simple Prolog programs (assuming a particular parallel execution model), but it is much harder to infer grain size for programs that use more complex algorithms involving recursions and complex data structures. Clearly, even for the cases where we cannot infer any grain size, we still need to use a dynamic and reasonable scheduling strategy.

9.2 A New Reconfiguring Heuristic

The intuition behind the strategy we describe in this chapter is that *workers 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, 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 to 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 during the execution. As we discussed previously in chapter 6, 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 [125, 150, 129, 33].

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.

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

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.

9.3 Description

The efficiency-guided strategy relies on previous observation of work performed. Workers are redeployed when they become inefficient. How do we know when a worker is inefficient? This is done by monitoring the time a worker is performing reductions during the execution time. The idea is simple: workers keep a record of the time they spend performing reductions. Every so often, the scheduler monitors whether the worker is keeping its efficiency. If the worker's efficiency is below a performance level threshold, the reconfigurer considers a redeployment to the worker.

The costs of redeployment are taken from the performance analysis of the Andorra-I system discussed in chapter 6. However they can also be given by the user.

The algorithm in Figure 9.1 presents the implementation of the efficiency-guided strategy. Idle workers try to be redeployed according to some possible lower cost redeployment. If the percentage of working time in a given period of time `time` is below the performance level threshold `eff_threshold`, the worker is redeployed. The function `get elapsed time` can get the time since the query started or from some definite time in the past. 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 from 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.

In algorithm 9.1, the variable `e` is the efficiency calculated for the worker. If the next lowest redeployment cost is to go to another team, a function to find the best team for the worker is used. The best team is the most efficient, i.e., the one that has spent more time working during the last period of time.

The thrust for this strategy is the observation that we need to keep workers with an efficient working ratio during execution time. So, we "rebaptise" efficiency as being:

$$e = r/ct \times 100$$

```
Begin
    working_time := Get my working time;
    time := Get elapsed time;
    e := (working_time × 100) / time;
    if e < eff_threshold then
      next_deployment := (curr_deployment + 1) mod worker_deployments;
      if ( next_deployment = change teams) or (next_deployment = change to slave) then
        Find best team t*;
        if there is no such team then
            return no change;
        else
            redeploy slave to team t*;
            return change to slave;
      return next_deployment;
    else
      return no change;
End.
```

Algorithm 9.1: MASTER OR SLAVE CALLS RECONFIGURER where 'r' is the total time the worker performs reductions with time, and 'ct' is the time elapsed since the last checking for efficiency.

The idea is similar to the one used by the Bristol scheduler to monitor when to release workers to the system, as described in section 5.3.2.

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 the application does not have too much parallelism, the performance of some workers will be below the efficiency threshold. Hence these workers are candidates to be released to the system. In other words, this strategy will solve two problems. First, allocation of workers to and- or or- tasks; this is done naturally as the workers will be becoming less efficient in their own teams and are redeployed to other teams. Second, releasing workers to the system; this is also naturally done, because workers that do not perform well for a very long time can be disposed of.

In conclusion, the system behaves like the following:

1) Every time interval, each worker will check if its average efficiency 'e' is above the threshold percentage T;

2) If $e > T$, keep worker in its team;

3) If $e \leq T$, redeploy worker to the level with cost immediately higher than the current level;

4) If the worker completed a whole cycle through the levels without having increased its efficiency, release it to the system. Here we might lose if the application suddenly creates a good amount of work, but we can not win all!

The problem of keeping workers doing efficient work most of the execution time is solved with this strategy, although some parallelism might be lost due to wrong redeployments at the wrong time. In any case this strategy is better than to redeploy workers depending on the size of work available if this size does not represent the actual work size.

It remains to solve the problem of *where* to migrate given that once the worker is redeployed to another level it may have several choices of places to move to:

- If a master is redeployed from or-work to or-work, it is up to the or-scheduler to decide what node to move to.
- If a master is redeployed from or-work to and-work, the reconfigurer finds the team that has more efficient workers.
- If a slave is redeployed from and-work to and-work, the reconfigurer finds the team that has more efficient workers.
- If a slave is redeployed from and-work to or-work, it is up to the or-scheduler to decide which node to move to.

9.4 Controlling Resource Utilisation

Bringing back the issue of taking and releasing workers to the system in order to utilise resources fairly in a multi-user environment, the efficiency-guided strategy could solve this issue naturally, as workers get overloaded with work or get inefficient during the execution.

The control of resource utilisation is one of the most important problems in parallel multi-user environments. The user cannot accept that a program runs fast on a dedicated machine, but gets extremely slow when more users are in the system sharing the resources to run other applications.

Most recent papers that discuss the problem of processor scheduling in shared memory multiprocessors tackle this problem. Most of them agree that allocating one process per processor is usually more efficient for parallel systems than to use time-slicing for several processes. From the point of view of the application, one process should be created per processor. Andorra-I employs this implementation technique. It remains to know what to do when more processes are created by other users than there are processors available.

Zahorjan and Cann [150] propose a dynamic strategy where the task-dispatcher at the application level releases processors dynamically according to the workload of the system and according to the parallelism available in the application. They produced simulated results for applications with different and varying degrees of parallelism during execution. According to their experiments, any attempt to reduce the processor releasing or processor allocating rates proved to incur too much overhead. They also proved their results experimentally on a Sequent Symmetry [90].

Tucker and Gupta [125] propose another strategy where the operating system leaves the user free from controlling resource allocation. They modified the Brown University Threads Package to contain code for controlling the number of runnable processes. They also implemented a server that communicates with the OS kernel to determine how many processes each application should have. The applications periodically communicate with the server to check how many processors they should have. Their assumption is the same as Zahorjan's: only one process is associated with a processor. They claim that the main reason for degradation in performance of parallel systems is the preemption of processes while they are executing within a critical section of the code. In such situation, other processes may be wasting processor resources just spinning, waiting to enter that critical section. Other possible reasons for degradation include the overhead of unnecessary task switching, and the problem of cache corruption when several processes are being multiplexed on a given processor.

It is important to note that every parallel multiprogrammed system (including SIMD architectures) suffers from this problem.

Beaumont [5] implemented in the Bristol scheduler a scheme that allow workers to be released to the system whenever the efficiency level of a worker drops below a threshold. Workers are allocated from the system when new or-work is created. He claims that for most of his benchmarks, adding fuel economy to Aurora resulted in a small slowdown of less than 5%, and saved around 10% of total computation time. For some of the programs the slowdown is much higher, but justifiable, since the reduction in total computation time is around 50%. Andorra-I also incorporates this feature to the system, although in a limited way. Workers are not allowed to be released while they are slaves, but a slave can wake up a master if some amount of and-work is created during execution.

We would like to have a version of the efficiency-guided strategy that releases and allocates resources dynamically. Because every processor is checking its efficiency every so often, it would be enough to check whether the processor's efficiency is below a certain very low threshold. If this is the case, the processor should be released to the operating system. As mentioned during this work, the key idea is to keep the algorithms as simple as possible. In fact, Zahorjan suggested several improvements to his algorithms that instead of improving efficiency, degraded the system even more. In order to allocate a processor, several parameters can be taken into account, like the workload of the machine, but we believe that a simple look at the efficiency of all workers allocated to the application would be a good index to decide when to allocate a new processor to the application.

Besides simplicity, we also need to keep workers doing useful work most of the time. In that case, the decision of releasing or taking another processor to the application should be a decision of the processor itself. In some scheduling algorithms, the decision is taken by a busy processor that signals to warn the system that more processors are needed or that a processor can be released. This approach presents two problems: one is that a busy processor deviates from its main job, that is doing useful work, to signal other processors, and the second problem is that there is no way of knowing how many processors the busy processor needs to signal. By leaving the decision of releasing a processor to the idle processor, the system is more distributed and makes quicker decisions.

9.5 Discussion

It might be argued that the approach of checking workers' efficiency may work well for slow machines as the Sequent Symmetry, where the overhead is hidden by the long time taken to execute a program. However the cost of reading a clock with the micro-second interface, and taking the elapsed time to calculate the percentage of time working has been shown to be negligible when compared to the execution time of the programs.

The overhead of scheduling with the efficiency-guided strategy is on average less than 1% of the total execution time. If the average execution time of the benchmarks is 3.127 seconds at 10 processors, even if we run the same benchmarks on a machine 10 times faster, the average execution time would be reduced to 0.3096 seconds (take 1% out of 3.127 and divide the result by 10: this would be the theoretical average execution time of the same benchmark set without any overhead of scheduling). If we add 1% of 3.127 to this value 0.3096, the program would run in a total execution time of 0.3406 seconds, on a machine 10 times faster at 10 processors. The average speedup that used to be 4.734 at 10 processors, would drop to 4.346 on a machine 10 times faster. Of course this would be the performance of the computation in the worst case, because the overhead of reconfiguring itself would be less in a faster machine, as the code would run faster.

This shows that the overhead of scheduling would be negligible for a set of benchmarks with a reasonable degree of parallelism.

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.

The issue of resource utilisation in multi-user environments is not implemented in the version of the efficiency-guided strategy we described in this chapter. It is definitely an important topic for future research.

9.6 Summary

In this chapter we described the *efficiency-guided* strategy to distribute and- and or-work among workers in Andorra-I. We described the strategy and its motivation, and presented its algorithm. The heuristic used by this strategy is to redeploy a worker whenever its performance drops below a threshold. By designing this strategy, we expected to overcome the drawbacks of the work-guided strategy by having a better heuristic to control frequency of task switching, and do better distribution of and-work and or-work. We would expect this new strategy to produce similar or better results than the work-guided strategy.

We started the chapter by giving the motivation to design a new reconfiguring strategy. We then described the strategy in detail, and defined the main redeployments for the workers. We established an order for the redeployments, based on the cost analysis performed in chapter 6. Next we described the algorithm for the strategy.

In this chapter we also discuss a very important subject, namely how to efficiently control resource utilisation in a parallel multi-user environment. The problem consists of allocating resources efficiently in order to eventually attend all users at the same time as keeping the maximum throughput possible.

Chapter 10

Results and Evaluation of the Efficiency-Guided Strategy

In the previous chapter we described the efficiency-guided strategy, an alternative to the work-guided strategy that instead of making decisions based on instant runtime information, makes decisions based on the past of the computation. The efficiency-guided strategy was designed because we are interested in achieving a target performance, and our work-guided strategy failed to achieve this goal for some of the benchmarks. As explained before, this happened because the work-guided strategy is short-sighted when making decisions.

In this chapter we study the performance of the efficiency-guided strategy described in the previous chapter, when compared with the work-guided strategy and target performances. In the first section of this chapter we evaluate the overall performance of the efficiency-guided strategy for our benchmark set. In the second section, we discuss in detail two of the benchmarks, analysing the reasons why the efficiency-guided strategy performs better than the work-guided strategy for the cases where the work-guided strategy did not perform similarly or better than our target performance. Some of the results shown in this chapter were presented before [44].

In the previous chapter, we discussed about the efficiency threshold. This is the parameter that regulates 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%.

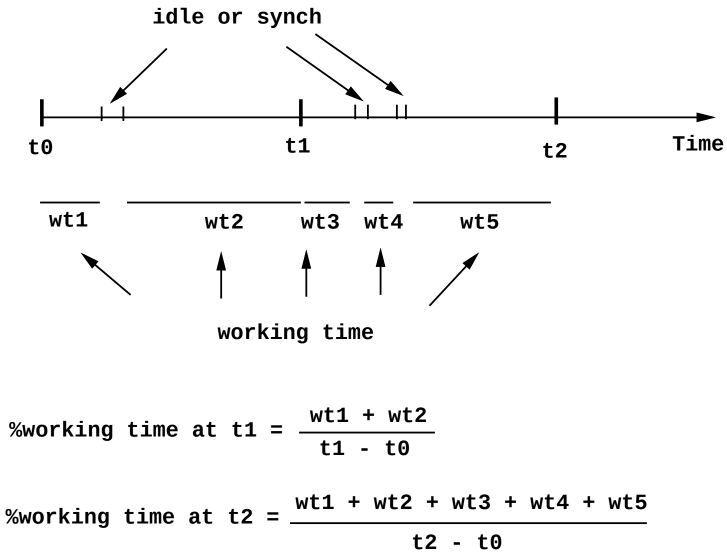


Figure 10.1: EFFICIENCY-GUIDED STRATEGY DURING THE EXECUTION TIME

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 uses always the time taken at the very beginning of execution as the beginning of the timing interval. Figure 10.1 shows how the system works with time. A worker checks its efficiency at time `t1`, where `t1` is the value of the clock. 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 did some experiments where the system forgets some of the past computation, however results turned out to be not so good as the results shown here. We believe that the reason is that each application needs a particular length of "window" to be able to perform well. If we use segments of time to check efficiency, and forget the past computation, we may ignore that some worker was inefficient in the past, and therefore, overall the load would not be balanced.

10.1 Performance Evaluation of the Efficiency-Guided Strategy

In this section 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 curves shown in chapter 8.

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 10.1: EFFICIENCY-GUIDED STRATEGY X WORK-GUIDED STRATEGY X BEST FIXED CONFIGURATION, AT 10 WORKERS

Table 10.1 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 10.1, is very similar to the overall performance of the work-guided strategy.

We showed in chapter 8 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 figures 10.2, 10.3, 10.4, 10.5, 10.6, we compare the efficiency-guided strategy with the work-guided strategy for up to 15 processors for some of the benchmarks. The curves in the graphs are differentiated by their thickness. The thick line shows the performance of the efficiency-guided strategy while the thin line shows the performance of the work-guided strategy.

At first sight, by observing the figures one can conclude that there is not much to choose between the two strategies. However, if we recall that the work-guided strategy did not outperform our target performance for some programs, and if we observe that the efficiency-guided strategy produces much better speedups for these kinds of programs, sometimes surpassing our target performance, we can conclude that the best choice is to use the efficiency-guided strategy.

For few cases (e.g., `flypan2` and `road_markings`, in figure 10.4), the efficiency-guided strategy seems to behave irregularly. We believe that this is due to some fine tuning we need to do to the efficiency-guided strategy. The implementation we have is not thoroughly tested, although yet produces very good results. For some of the benchmarks that present predominantly and-parallelism, the efficiency-guided strategy drops slightly in performance when compared with the work-guided strategy. The main reason for that is that as the efficiency-guided strategy is conservative and takes time since the query

Figure 10.2: EFFICIENCY-GUIDED STRATEGY: ONLY OR AND ONLY AND BENCHMARKS

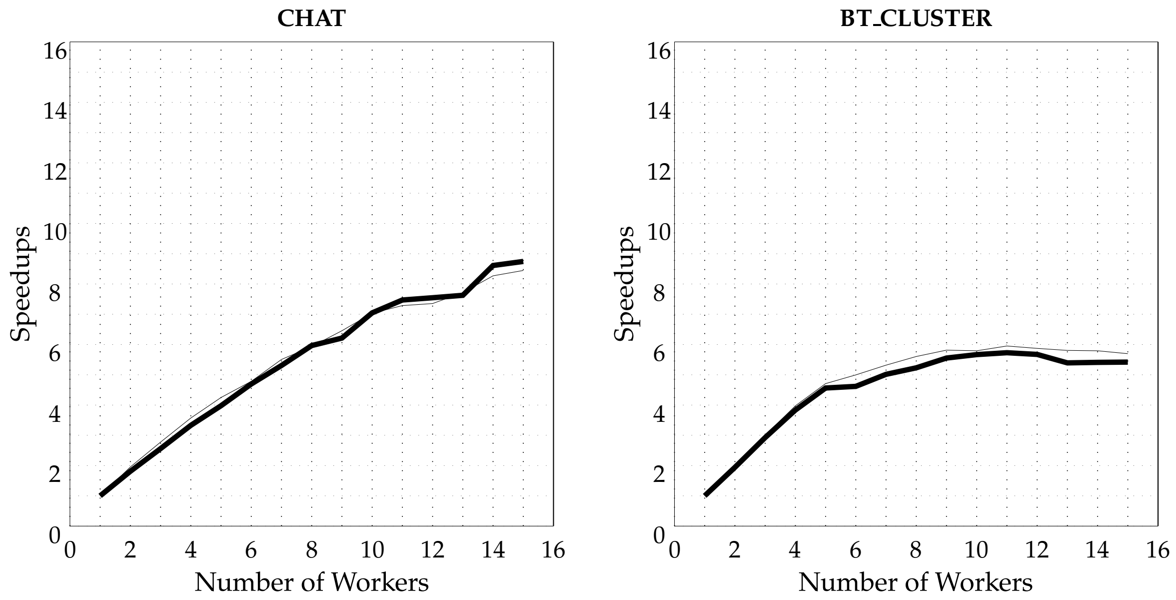
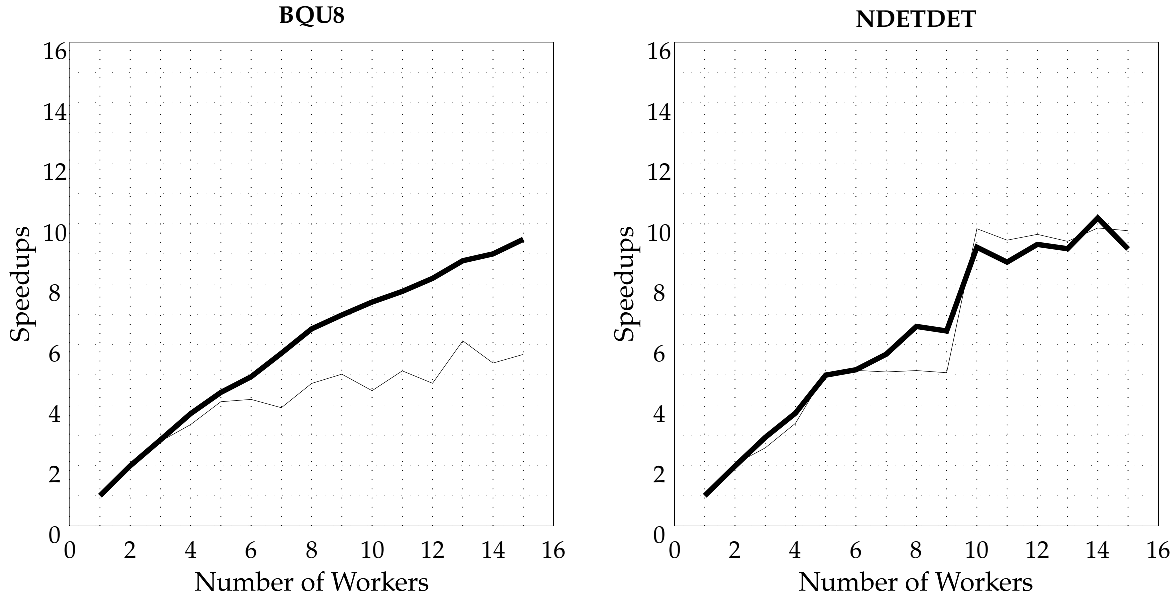


Figure 10.3: EFFICIENCY-GUIDED STRATEGY: HIGH OR, LOW AND BENCHMARKS

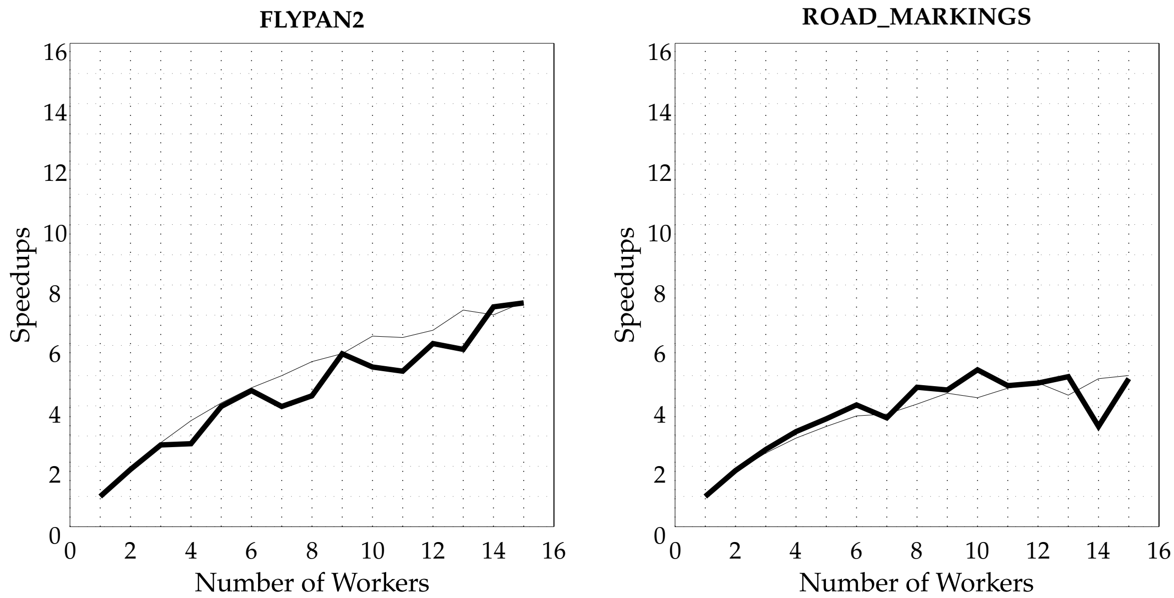


Lines are differentiated by their thickness. The thin line corresponds to the work-guided strategy. The thick line corresponds to the efficiency-guided strategy.

started, workers may take a long time to realise they need to be redeployed elsewhere. By the time they realise they need to change, the execution finishes. However the decrease in performance for these few cases is negligible when compared with the increase in performance achieved by the other programs. It is important to notice that our objective is to perform well for most cases, and that it is impossible to find a single strategy that produces optimal results for all cases.

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.

Figure 10.4: EFFICIENCY-GUIDED STRATEGY: HIGH AND, LOW OR BENCHMARKS



10.2 Case Studies

In order to better illuminate the difference between the two strategies: efficiency-guided and work-guided, in this section we study their performance in more detail by showing how Andorra-I 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 are shown in figure 10.3.

Figure 10.5: EFFICIENCY-GUIDED STRATEGY: LOW OR, LOW AND BENCHMARKS

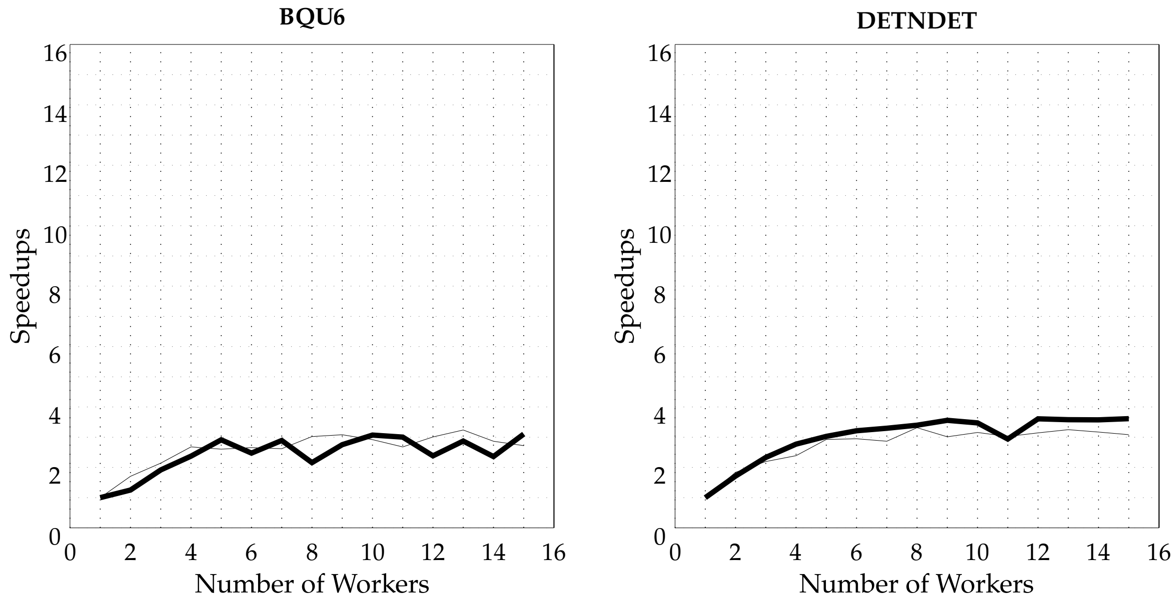
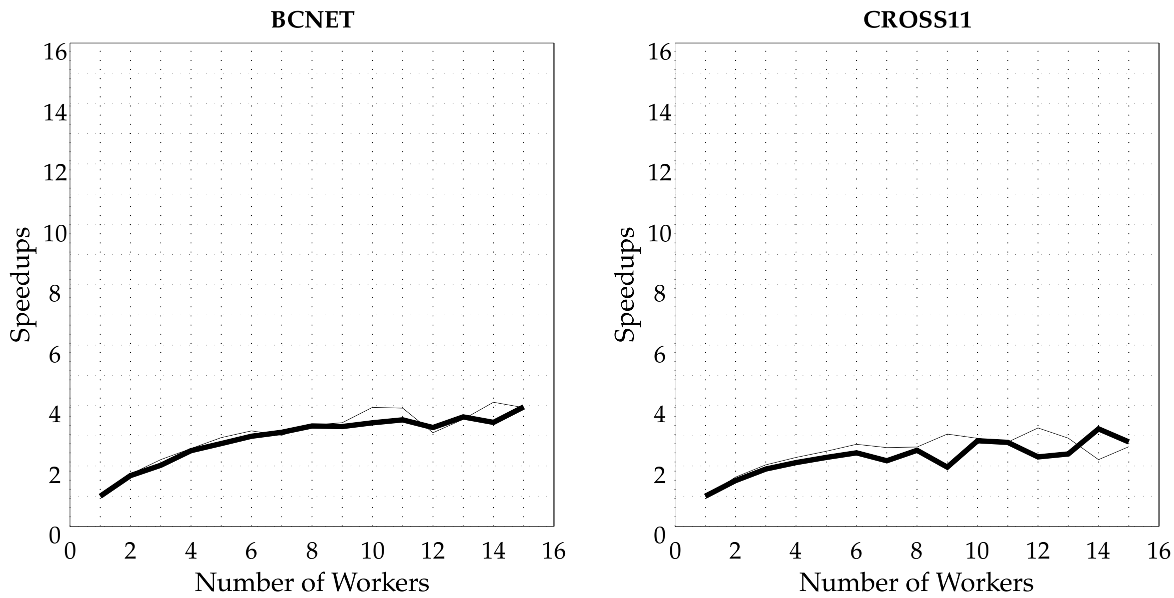


Figure 10.6: EFFICIENCY-GUIDED STRATEGY: LOW AND, LOW OR BENCHMARKS



10.2.1 Program `bqu8`

The first benchmark, `bqu8`, presents a significant amount of or-parallelism with little and-parallelism.

As is clearly shown in figure 10.3, the speedups using either strategy start to diverge above 6 workers. When we increase the number of processors, the efficiency-guided scheduling strategy scales better than the work-guided strategy allowing an improvement in average speedups of about 44% over the work-guided strategy, from 6 to 15 processors. This happens because the work-guided 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-guided 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-guided strategy applies almost no reconfigurations in 10 runs with 15 processors, the work-guided strategy produces 517 reconfigurations! (for the best run out of 10). So, the efficiency-guided strategy also refrains workers from doing unnecessary reconfigurations. Although sometimes the speedup is improved by a bigger number of reconfigurations, this is not a rule.

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, as discussed previously in chapter 6, 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 10.7 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.

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 efficiency-guided strategy (the total overhead is less than 0.2% of the total execution time). 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-guided strategy, the time spent in and-scheduling is higher than the time spent in and-scheduling when using the efficiency-guided strategy. This happens because there is little and-parallelism in this program, but with the work-guided strategy some of the slaves are running out of and-work after being reconfigured as slaves. On the other hand, the time spent in or-scheduling, when using the efficiency-guided strategy, was greater than the time spent in or-scheduling when using the work-guided strategy. The ratio between the total time spent looking for and-work with the work-guided 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-guided strategy than with the efficiency-guided strategy. Although the time spent in or-scheduling is higher for the efficiency-guided strategy, this strategy produces better speedups than the work-guided strategy. In other words, workers become inefficient whenever they are working as slaves, because there is very little 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 10.7, we can observe that, with the efficiency-guided 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-guided strategy. The bottom left graph in figure 10.7 shows how workers have their idling time reduced compared with the work-guided 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 to this variation is synchronisation between workers (e.g., locking and waiting for work).

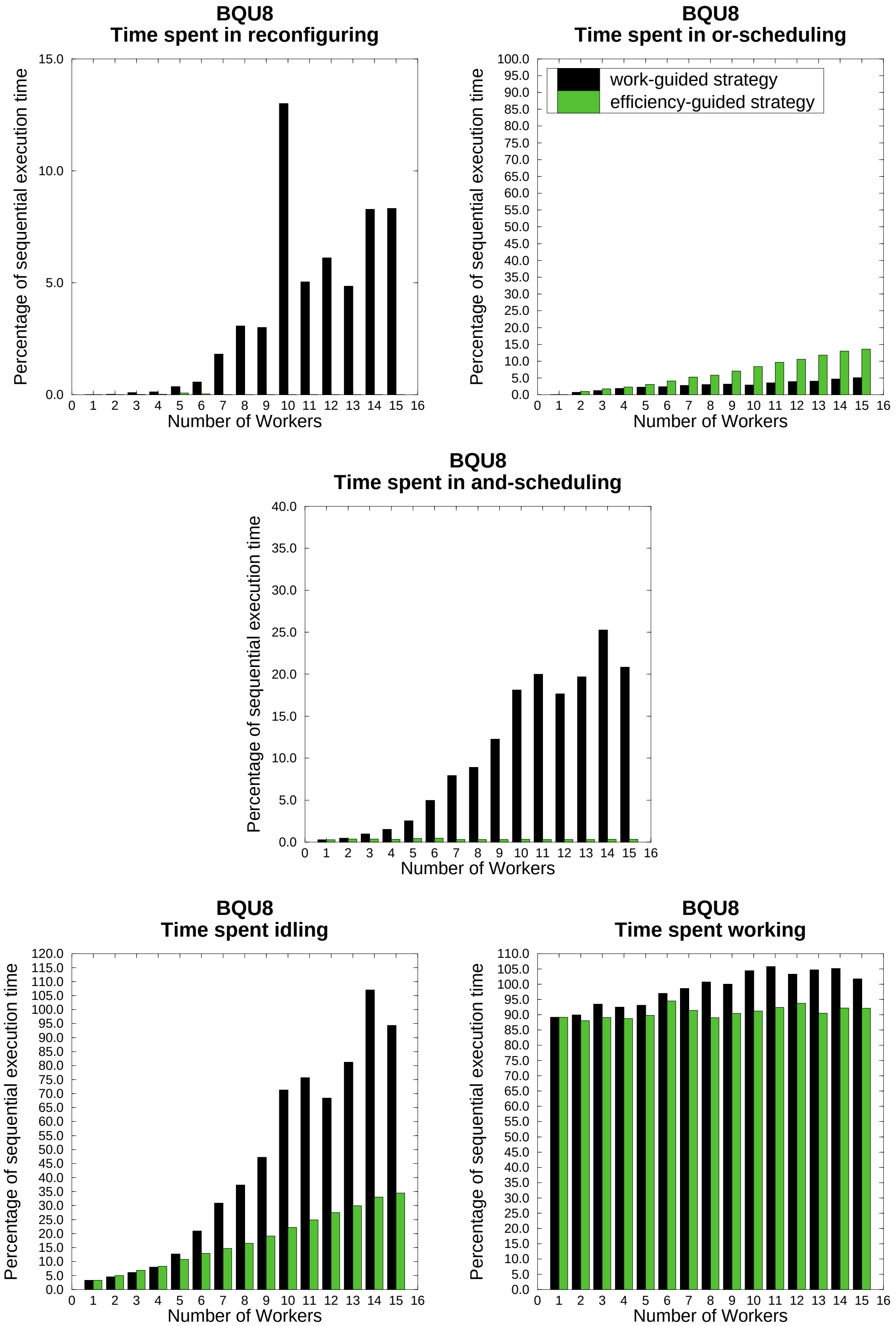


Figure 10.7: `bqu8`:ACCUMULATED TIME SPENT ON SEVERAL ACTIVITIES

10.2.2 Program `ndetdet`

The 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, on 10 processors). The ideal tree for this program is shown in figure 10.8. A choicepoint is created with 10 branches. Each of these 10 branches create another choicepoint 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.

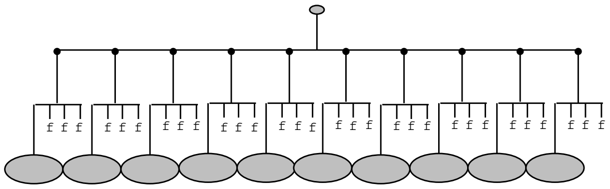


Figure 10.8: IDEAL ANDORRA-TREE FOR `ndetdet`

Assume that we can execute this program only in or-parallel for the second part of the tree (i.e., after the first small determinate phase) using exactly 10 processors in a static configuration where each processor will be responsible for executing one branch. Ignoring the branches that will fail very quickly (they produce shallow failures), assume that the time to compute that part of the tree in or-parallel is `T`. Thus `T` is the sequential time to execute each branch without having any and-parallelism being exploited. Provided that we have and-parallelism in each branch (which we actually have), only the static configurations of 10 workers with 5 masters or more produce maximum speedups of 10 on 15 workers. At least 5 teams will divide the computation equally in order that the total parallel execution time is T. For these kinds of configurations, ignoring any parallel overheads, we have a parallel time for the or-parallel phase of `T`. This means that the best theoretical speedup with a static configuration for 15 workers is 10 (the total sequential time is `10T`).

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

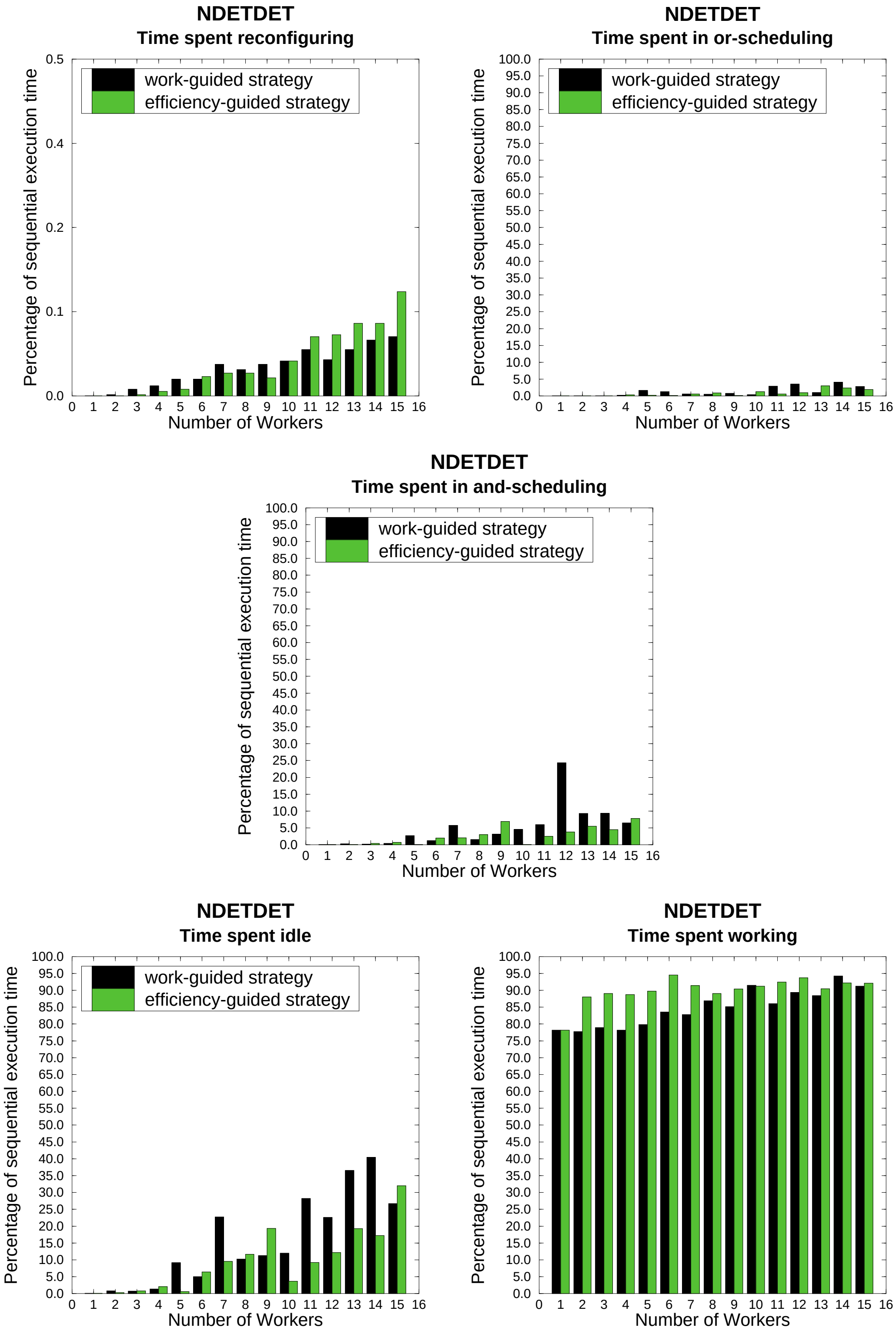


Figure 10.9: `ndetdet`:ACCUMULATED TIME SPENT ON SEVERAL ACTIVITIES

strategy, unless we limit the execution of a worker by the depth of the computation[1]. If we consider the slaves working full time in each branch the speedup is still 10. Figure 10.3 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-guided 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-guided strategy begins to outperform the work-guided strategy. The efficiency-guided 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 10.9 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.

10.3 Summary

In this chapter we presented results comparing the efficiency-guided strategy with the work-guided strategy. It seems that the heuristic used by the efficiency-guided strategy overcomes one of the drawbacks of the work-guided strategy by producing better results for the programs that have more or-parallelism with fine grained and-parallelism on the branches.

For the rest of the benchmarks, the efficiency-guided strategy performs similarly to the work-guided strategy, except for a few cases, where the performance slightly drops when compared with the work-guided strategy. However the increase in performance for those cases where the work-guided strategy was not doing so well is very significant when compared with the small decrease in performance incurred in some benchmarks. It is important to stress that it is not possible to find a single strategy that achieves the best performance for all benchmark set without some cost.

Overall the results showed that the efficiency-guided strategy is preferable to be used

[1]Thanks to David H. D. Warren for pointing me out this possibility.

for three main reasons: (1) it improved significantly the performance of a given class of applications that contains initially or-parallelism, and fine grained and-parallelism on the or-parallel branches; (2) it naturally limits the frequency of task switching in the system; (3) the increase in performance for some of the computations is quite high when compared with the decrease in performance for some other computations.

Therefore the new heuristic used by the efficiency-guided strategy was proved experimentally to be better than the work-guided strategy.

Chapter 11

Conclusions and Further Work

In this chapter we summarise the objective of this work, and identify the main conclusions and contributions. We also discuss immediate steps forward and long term further research.

11.1 Conclusions and Contributions

Parallel logic programming systems seem to be the next step towards faster implementations of logic programming systems. It brings several new challenges to the system implementor, one of them being the problem of efficiently distributing processors between and-work and or-work which arise naturally during execution of the programs. This is an important and hard problem to solve in the scope of parallel logic programming systems.

The main objective of this work was to give a solution to the problem of efficiently distributing processors between both kinds of work, in the scope of Andorra-I. As Andorra-I aims at exploiting parallelism implicitly, we were challenged to make the system distribute and- and or-work in a way transparent to the user. Our work was of great practical importance for Andorra-I, because we added a key missing component, the *reconfigurer*, responsible for doing *automatic* distribution of and- and or-work among the processors.

Before our work, Andorra-I depended upon a fixed configuration of workers into teams, which had to be selected by the user. This caused three main drawbacks:

- The user had to manually adjust workers into teams to run the programs. This is a difficult and inconvenient task, especially in the light of the Andorra-I aim to exploit parallelism implicitly.

- In most cases, the user would be unlikely to be able to choose the best fixed configuration, leading to performance far from optimal.
- Even if the user could (somehow) choose the best fixed configuration, that would often produce performance below the best possible (with dynamic reconfiguration).

In our work, we studied the problem of automating the distribution of workers between and-work and or-work, and showed the first known solution for this problem. Because we wished to obtain speedups close to the best possible speedup, we investigated two different reconfiguring strategies, the *work-guided* strategy and the *efficiency-guided* strategy.

The work-guided strategy relies on assessing the amounts of work existing in the teams or in the choicepoints to decide when to redeploy a worker. The heuristic behind this strategy is to attract workers to places with more work to be done. Assuming we can accurately estimate the amount of and-work and amount of or-work available during the execution, this seems to be a reasonable heuristic.

The other strategy, the efficiency-guided strategy, makes decisions by monitoring the efficiency of a worker, where efficiency is measured as percentage of time doing useful work over execution time. As the objective of every parallel system is to keep processors busy most of the execution time, the efficiency-guided strategy also seems to be a reasonable heuristic.

In order to keep the reconfigurer overheads low, we designed very simple decision algorithms. In order to control the frequency of reconfiguring we designed our strategies around guideline parameters obtained from a performance analysis of Andorra-I.

We tested both strategies with an extensive set of experiments. The results support the strategies proposed by showing that both strategies can be implemented efficiently, with small overheads. They confirm that both reconfiguration strategies achieve a substantial overall performance improvement over any plausible fixed team version of Andorra-I. As regards whether the two reconfiguring strategies achieve optimal performance, results show that for programs containing a varying balance of and- and or-parallelism, both strategies achieve good speedups, better than the speedup achieved with any fixed configuration. For most applications, the work-guided strategy works better than the best fixed configuration, but for some cases it does not achieve this target. The efficiency-guided strategy seems to be a general solution for dynamically scheduling and- and or-parallelism in parallel logic programming systems, since it produces similar or better results than the ones produced with the best fixed configuration, for the benchmark set tested.

To the best of our knowledge, Andorra-I is the first parallel logic programming system that exploits both dependent and-parallelism and or-parallelism. With the help of our work it became a fully practical system that runs real applications efficiently.

We can summarise the main contributions of our work as being:

- We tackled a novel problem in the context of parallel logic programming systems namely how to distribute and-work and or-work automatically among the processors.
- For Andorra-I, in particular, we added a key component to the system, the *reconfigurer*, to automate the distribution of and- and or-work. This piece of software removed from the user the difficult business of choosing a fixed configuration of workers to run a particular computation.
- We proposed two reconfiguration strategies, both of which showed good performance (better overall than any plausible fixed configuration and close to our target performance).
- We believe that the technique of guiding the distribution of resources based on monitoring efficiency of resource utilisation is novel and could be applied to parallel systems other than logic programming systems, in the context of automating the exploitation of parallelism.

11.2 Further Work

Both strategies for the reconfigurer work reasonably well with a wide range of applications containing varying degrees of parallelism. However, as is always the case, there is still scope for improvements.

11.2.1 Immediate steps forward

One immediate step forward will be to correctly modify the synchronisation algorithms in Andorra-I in order to allow slaves to leave their team during a determinate phase. The results presented in this thesis were produced with a version of the system that contains this restriction. The engine does not call the reconfigurer during the determinate phase, which means that idle slaves cannot be redeployed until the determinate phase finishes. Our results are already very good, but we believe we can improve the execution of some applications by relaxing this restriction, and therefore, allowing the reconfigurer to exploit parallelism to a full extent. Applications that have many alternatives and very long determinate phases will particularly benefit from this modification.

11.2.2 Resource Utilisation in Parallel Multiprogrammed Environments

A second immediate step is to request processors from the system on demand instead of allocating all processors at once at the very beginning of the execution. Currently Andorra-I forks as many processes as there are processors available, and the DYNIX operating system guarantees each process is allocated to one processor. Problems arise when the system needs to time-share processors among several users tasks. As most of the current parallel and distributed operating systems do not have enough "intelligence" to attend to the demand of the parallel user applications, this needs to be dealt with by the parallel system programmer.

Multi-tasking in parallel systems affects directly the end-user, because execution times can increase significantly when several users' applications are competing for the same resources. For example, if we have several Andorra-I users running different applications at the same time using all resources, the throughput falls greatly. We believe that a strategy that includes non selfish utilisation of resources by the applications can improve the overall performance of the multi-user environment. The reconfigurer has embedded in its design the possibility of dealing with this problem, first by releasing a processor to the system when it is inefficient after some period of time, and second by taking a resource from the system when the system starts to be overloaded by the parallelism available. Several strategies are possible, but the main key point is to keep algorithms as simple as possible in order to avoid excessive overheads.

11.2.3 Compile-Time Granularity Analysis

We believe that the reconfiguring strategies proposed in this thesis can be combined with information collected at compile-time in order to improve the quality of the decisions to reconfigure.

The main reason why we are interested in using compile-time granularity analysis combined with some dynamic strategy is that static scheduling alone and dynamic scheduling alone have general disadvantages. As it is known from Lewis and Rewini [81], the major disadvantage of static scheduling in parallel systems in general is its inadequacy in handling branches and loops in the context of programs written in imperative languages. In the context of parallel logic programming systems, the disadvantage of static scheduling is in handling recursions and the irregularity of the amount of parallelism which arises during a computation. The major disadvantage of dynamic scheduling is its inadequacy in finding global optima and the corresponding overhead which arises because the schedule must be determined while the program executes. When the dynamic algorithms make decisions based on static information, we can combine the advantages of static scheduling with dynamic scheduling and overcome the disadvantages of having either kind of scheduler.

Compile-time analysis would help to produce general information such as places where the computation has a probability of being sequential, sizes of and-work and or-work for some predicates and amount of potential speculative work in the tree. Recent work done by Lin and Debray [82] and followed by García and Hermenegildo [49] reports on an implementation of a system that automatically generates simple equations, derived from recurrence equations, that can be used at runtime to help scheduling decisions. This adds a new overhead to the reconfigurer algorithms we presented here because, before doing any "guessed" decision, the reconfigurer will try to rely on the information obtained from the compile-time analysis. This information consists basically of a simple equation that gives the size (in number of inference steps) of a goal or of an alternative. Doubtless, if this information is precise enough, any scheduler would benefit from it. However the results published are still far from some useful information that can really be applied to help the reconfigurer. The system developed by Lin treats only very simple cases. We would like to emphasize that although we believe that compile-time analysis can help scheduling decisions, we do not believe that a parallel system can rely **only** on static information obtained from compile-time analysis. In that case, totally dynamic decisions need to be done anyway.

11.2.4 What about Speculative Work?

As defined in chapter 6, unnecessary work is any kind of work that is done in parallel that would not be done sequentially. In other words, it is extra work compared with the amount of sequential work. *Speculative* work is then the amount of work that may not be necessary to perform. The execution of speculative work may reduce the total parallel execution time, if at runtime the speculative work is proved to be necessary. Many parallel systems adopt the approach of allowing workers to execute speculative work, in order to speed up the parallel execution at the cost of increasing the amount of work performed. In parallel logic programming systems this corresponds to increasing the search space with respect to the search space of the sequential model.

In parallel logic programming systems we can have two sources of speculative work: and-speculative work and or-speculative work. Or-speculative work appears in the scope of cuts and commits. Clauses to the right of a clause that contains a cut are considered to be or-speculative work. Clauses to the right and to the left of a clause that contains a commit are considered to be or-speculative work. And-speculative work arises from other goals in a conjunction if one of the goals in that conjunction is likely to fail.

In Andorra-I, the amount of or-speculative work is inferred by the or-scheduler through cut counters and cut scopes information generated at compile-time analysis [66, 8]. Andorra-I (and any known existing and-parallel logic programming system) does not incorporate any feature to detect and-speculative work. Sophisticated compile-time analysis needs to be done in order to detect speculativeness of and-work.

Dealing with speculative work in parallel logic programming systems is still an incompletely solved problem. Some schedulers have tackled the problem (e.g. the Bristol scheduler [8] and the Dharma scheduler [112]) in the context of or-parallel logic programming systems. Recently, the Muse system also incorporated to its or-scheduler the ability of dealing with speculative computations [2]. A solution for Parlog, requiring help of the programmer, is reported by Gregory [55].

Many issues need to be taken into account when dealing with speculative work. What kind of basic algorithm is going to be used by the system, e.g. always do mandatory work or allow workers to choose some speculative work? If the system is allowed to select speculative work, should it always seek better work? If so, how often? How can we classify speculative work? How can we infer that work is always mandatory or always unnecessary? And other questions.

11.2.5 Application to other Parallel Logic Programming Systems

The strategies to distribute and-work and or-work described in this thesis served to improve the execution of Andorra-I. Moreover, the utilisation of a reconfigurer is fundamental to achieve better performance in a system that exploits both and- and or-parallelism. It appears that the ideas presented in this thesis could be extended to other systems such as EAM [140], ACE [58], PBA [63], IDIOM [62], ParAKL [92], Prometheus [110], and Penny [91] that aim at combining and- and or-parallelism. These systems will sooner or later face the same or similar problems as the ones faced by the Andorra-I system.

Although the and-parallelism exploited by these models is different from Andorra-I, we believe that the same ideas could be used. ACE, as mentioned before, aims at exploiting independent and-parallelism as in &-Prolog and or-parallelism as in Muse. The system, similarly to Andorra-I, is designed in a way that there are teams of workers to execute and-parallel goals and teams cooperate to exploit or-parallelism. As in this model the and-parallelism is coarser grained than in Andorra-I, the choice to redeploy a worker to a piece of and-work needs to be different from the one used in Andorra-I. It even may be the case that a random choice will work well, but this needs to be proved experimentally.

IDIOM has three computation phases: a determinate phase, an independent and-phase, and a non-determinate phase. The distribution of work could be done by choosing to redeploy a worker to one of the three kinds of work available. Depending on the study of performance of the system, it may well be that we need to add another variable to the reconfigurer algorithm in order to decide among three kinds of work, instead of only two as in Andorra-I. But this also needs to be proved experimentally.

Penny, a system based on the Extended Andorra Model, and that implements a parallel version of AKL, also creates two kinds of work during execution from the and-boxes and or-boxes. Their implementors claim that, unlike Andorra-I, there is no need for having separate schedulers for and-work and for or-work, with a top reconfigurer. We claim that the design of any system needs to follow some structuring, take advantage of the techniques already efficiently implemented, and furthermore keep decisions as distributed as possible to maintain good locality. In that case, these functions: finding and-work, finding or-work, and deciding between and-work and or-work should be kept separated. In addition, as we believe that exploitation of and-parallelism and exploitation of or-parallelism have different costs, it may be interesting to keep search of either kind of work separated.

Some of these systems do not use the concept of teams of workers as Andorra-I. However the techniques discussed in this thesis do not need to assume teams of workers. In fact, in any system where we can classify kinds of work, for example: and-work and or-work; low cost work, high cost work; bottom-most work, top-most work etc, we can apply any of the two ideas proposed in this thesis.

We believe that future logic programming systems are going to exploit more than one form of parallelism with highly sophisticated global analysis techniques to generate as precise information as possible to allow the systems to exploit all parallelism possible. Dynamic reconfiguring of workers for these systems becomes fundamental, since different sources of work appear at different execution times. Also because logic programming applications produce very irregular computation patterns, systems need to adapt quickly to the change in amounts of parallelism existing.

Although a parallel logic programming system may not have much potential for truly massively parallel processing, because most of the logic programming applications have a very irregular computation, it is certainly capable of achieving very good speedups in machines of modest size, say from 4 to 64 processors. With the arrival of multiprocessor workstations to the market, the goal of exploiting parallelism implicitly, and the declarativeness offered by logic programs, parallel logic programming systems will certainly be an alternative for the conventional parallel systems currently existing.

Appendix A

The Reconfigurer Interface Macros

This is a set of the main interface functions used by the reconfigurer. The interface also contains functions for collecting statistics.

Here, and in the remaining of the text, we use the prefix `Top_Sched` for macros of the reconfigurer, `And_Sched` for macros of the and-scheduler, `Sched` for macros of the or-scheduler and no prefix for macros defined by the engine.

- Engine → Reconfigurer:
  - `Top_Sched_Init(int argc, **argv)`

    It is responsible for initialising the reconfigurer, building the worker and team structure, and updating some variables that are important to the engine.

    It is called by the engine, before forking.
  - `Top_Sched_Find_Me_Work(TO_MASTER, TO_OTHER_TEAM)`

    It is called whenever no and- work is found for a slave, in a team. The reconfigurer will redeploy the worker to another team or change it to a master.
  - `BOOL Top_Sched_Find_Remote_Work()`
  - `BOOL Top_Sched_Slave_Check_Efficiency(`
    `int wid, TO_MASTER, TO_OTHER_TEAM))`

    Returns a true boolean value if there is some and-work to be done in another team. It is called by the engine when no and-work is found for a slave in a team, as in the previous function.
  - `Top_Sched_Initial_Configuration(`
    `int team [MAX_TEAMS][MAX_WORKERS])`

Return to the engine the initial configuration of the teams, in the 'team' array variable. The position [i][0] of the array corresponds to the number of workers in the team. The array is arranged in a way that the masters come first in the sequence. The identifier of each team corresponds to the identifier of the master in the team.

  - `int Top_Sched_Total_Workers()`

    This function is called by the engine to know the total number of workers in the system.

  - `int Top_Sched_Total_Teams()`

    It returns to the engine the current number of teams in the system.

  - `int Top_Sched_Maximum_Teams`

    It returns to the engine the maximum number of teams in the system (for a flexible reconfigurer, it corresponds to the total number of workers, because every worker is a potential master and can be boss of a team. For other reconfigurers, this number can be different, if some workers are always kept as slaves).

  - `BOOL Top_Sched_Is_Master(int worker_id)`

    It returns to the engine a true boolean value if the worker is a master.

  - `BOOL Top_Sched_Is_Slave(int worker_id)`

    It returns to the engine a true boolean value if the worker is a slave.

  - `Top_Sched_Check()`

    It is called from time to time (for instance, every ten Andorra-I calls) and can change the arrangement of teams based on global information in the tree. It is not defined yet. It is useful to wake up some workers or put some workers to sleep.

- Engine → Or-Scheduler:

  All the macros supplied by the or scheduler interface (see [1]). Slight modifications were done to some of the functions of the or- scheduler (Sched_Start_Work, Sched_Die_Back etc) to adapt them to run with the reconfigurer. A way out to the reconfigurer should be introduced in each or-scheduler macro whenever a worker runs out of or-work. The or-scheduler macros should be prepared to accept that a worker changes from master to slave. A master worker should be allowed to become a slave on initialisation (if no or-work is found in the very beginning of computation when calling the or-scheduler macro Sched_Start_Find_Work).

  A new function is introduced:

  - `Sched_Restart_Work(HALTED, CHANGE_TO_SLAVE)`

    It is to be called by a new master, i. e., a slave worker that became a master.

- Engine → And-scheduler:
  - `And_Sched_Init(int argc, **argv)`

    This macro is called by the engine in order to initialise the and-scheduler.
  - `And_Sched_Find_Work`

    It is responsible for looking for work in the run queues inside the same team.
- Reconfigurer → Engine:

  There is no difference between these two macros. They have different names just to make the reconfigurer algorithms clearer and more readable.
  - `Redeploy_Slave_To_Team(int team_id)`

    This macro informs to the engine that it has to be redeployed from one team to another.
  - `Redeploy_Master_To_Team(int team_id)`

    This macro informs to the engine that it has to be redeployed from one team to another. Unlike the previous function, this macro is called when a master became a slave.
- Reconfigurer → And-scheduler:
  - `int And_Richness(int worker_id)`

    This macro is called by the reconfigurer to get information about the amount of parallel and-work available in the run queue of goals of `worker_id`.
  - `BOOL Empty_Run_Queue(worker_id)`

    It is called in order to know if the pool of goals of `worker_id` is empty.
- Reconfigurer → Or-scheduler:
  - `int Sched_Public_Richness(int master_i)`

    This macro is called by the reconfigurer to get information about the public richness (amount of global parallel or-work) of a master worker.
  - `int Sched_Private_Richness(int master_i)`

    This macro is called by the reconfigurer to get information about the private richness (amount of local parallel or-work) of a master worker (only provided by the Dharma scheduler at the moment).
- Or-Scheduler → Reconfigurer:
  - `Top_Sched_Change_Master_To_Slave()`

  - `Top_Sched_Master_Check_Efficiency(int wid)`

    It is called whenever a master runs our of or-work on backtracking or on starting work.

- Or-scheduler $\rightarrow$ Engine:
  - all supplied by the engine-scheduler interface (see [120]).

Appendix B

Work-Guided Strategy: Variation of Parameters

In order to investigate further the behaviour of the Andorra-I system under the work-guided strategy we ran programs for several numbers of processors with some different parameter settings. Graphs numbered from B.1 to B.6 compare speedups between the fixed configuration and several versions of the work-guided strategy with varying parameter values.

We tested the work-guided strategy up to 15 processors for all benchmarks by using the following choices of parameters:

- choicepoint-guided, with and-cost of 10, or-cost of 8, and/or-threshold of 3, or/and-threshold of 3, assuming one goal produces 1 reduction and each choicepoint produces only one alternative. This is a very conservative choice of parameters (*work-guided strategy (1)*);
- the same strategy before with the same parameters, but using a better estimate for the amount of or-work: number of alternatives left on the execution tree (*work-guided strategy (1-alt)*);
- still using number of alternatives as an estimate for the size of or-work, but using parameters taken from the performance analysis of Andorra-I: and-cost of 3, or-cost of 7, and/or-threshold of 0, or/and threshold of 0, assuming each goal produces 10 reductions and each alternative produces 5 reductions (*work-guided strategy (2)*). The number of reductions produced by each goal and each alternative were arbitrarily chosen. Results can vary for a different choice of values for these parameters.

Figure B.1: WORK-GUIDED STRATEGY: ONLY OR BENCHMARKS

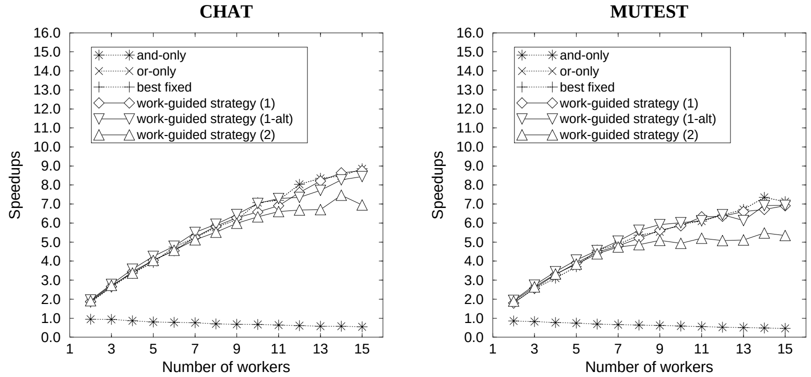


Figure B.2: WORK-GUIDED STRATEGY: HIGH OR, LOW AND BENCHMARKS

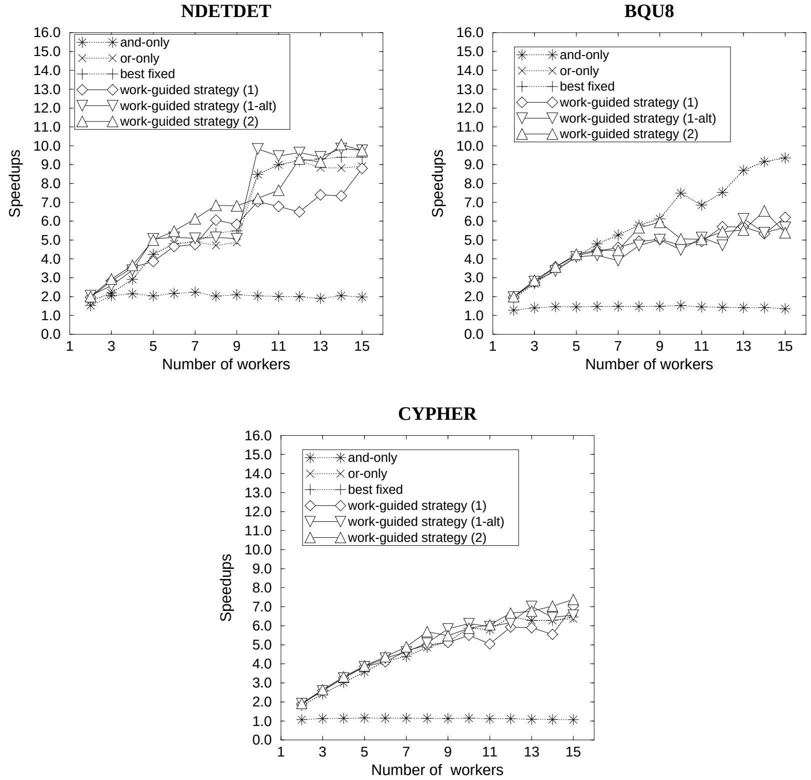


Figure B.3: WORK-GUIDED STRATEGY: HIGH AND, LOW OR BENCHMARKS

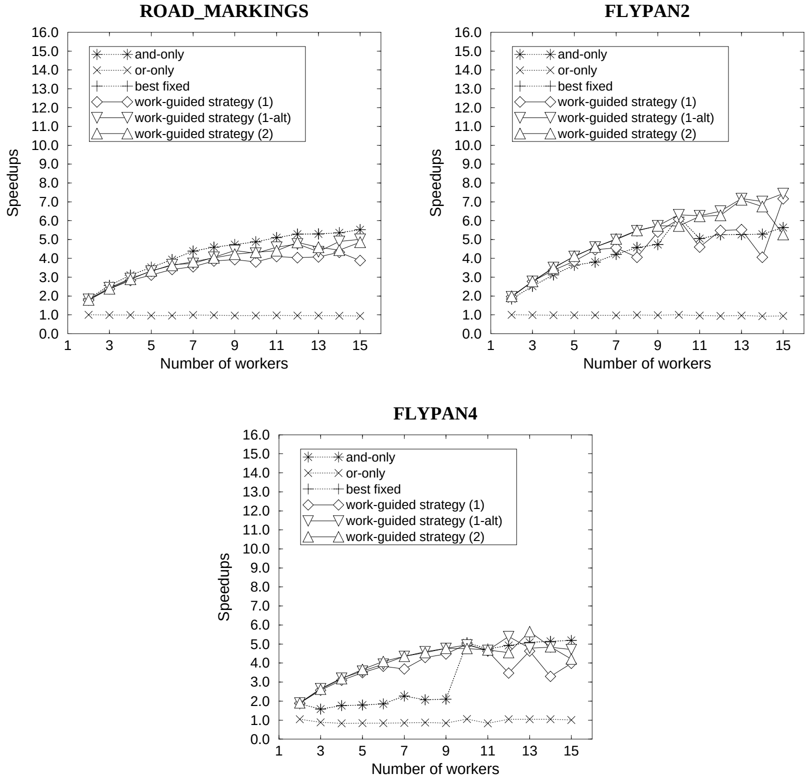


Figure B.4: WORK-GUIDED STRATEGY: HIGH OR, HIGH AND BENCHMARK

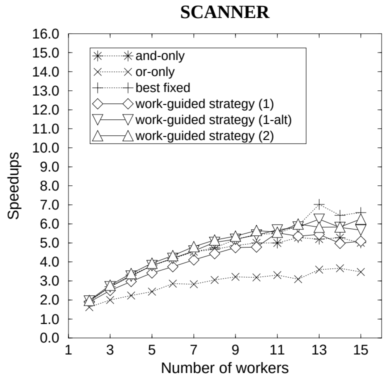


Figure B.5: WORK-GUIDED STRATEGY: LOW OR, LOWER AND BENCHMARKS

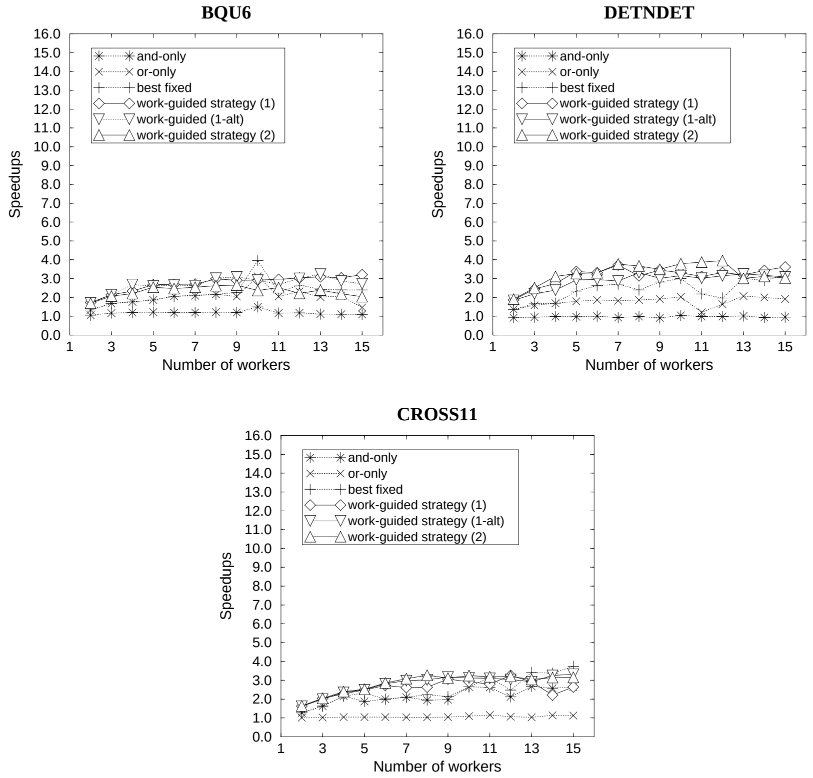
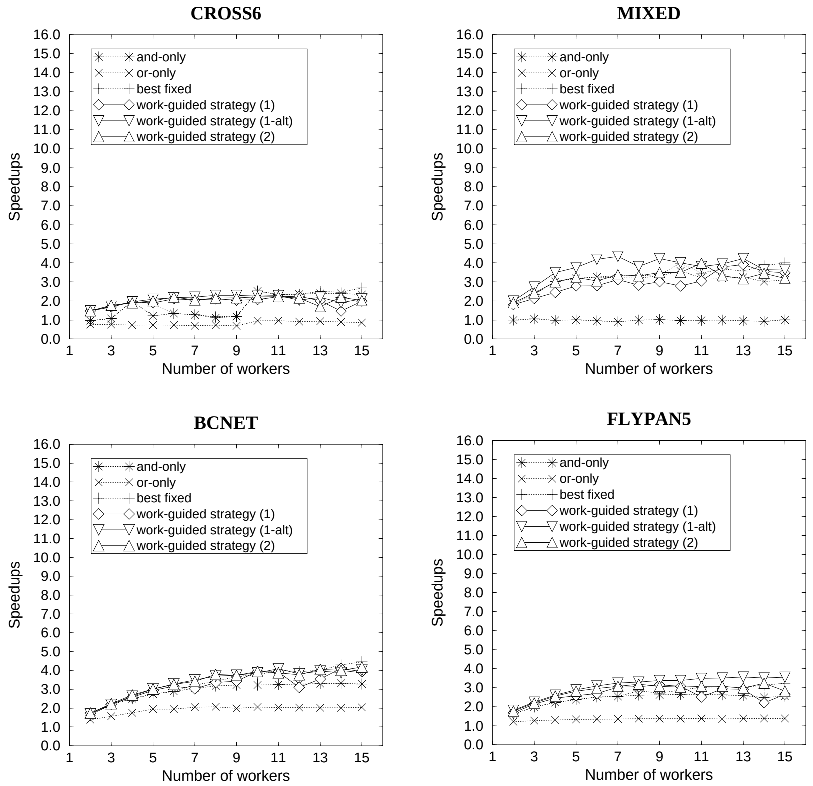


Figure B.6: WORK-GUIDED STRATEGY: LOW AND, LOWER OR BENCHMARKS



The programs that present more or-parallelism are shown in the first 5 graphs. Generally, the best average speedup for each of these programs is achieved for the fixed configuration that presents the most number of masters. The and-parallel benchmarks achieve better speedups with more number of slaves. Some fixed configuration curves (and-only, or-only and target) are shown in order to give a rough idea of what kind of parallelism the benchmark has, and if the combination of both kinds of parallelism result in a improvement in performance. The best fixed configuration curve (our target performance) shows that some of the programs can improve the execution speed by combining and-parallelism with or-parallelism. This is shown for programs `bqu6`, `detndet` and `cross11`, in figure B.5, where the best fixed configuration achieves results better than exploitation of or-parallelism only (or-only curve) and exploitation of and-parallelism only (and-only curve). The reconfigurer *automatically* achieves similar or better results than the target performance with different parameter settings.

Overall, the best parameter setting is the one used for work-guided strategy (1-alt). This is shown to be the best, because workers are more restricted to be redeployed to and-work than to or-work due to the and-cost and or-cost values. In Andorra-I, the more slaves we add to a team that does not have enough parallel and-work to be done, the more inefficient the system becomes. In that case, if we add a mechanism to the system to avoid giving preference to and-work, this would result in better performance, in general. This is achieved by work-guided strategy (1-alt), because we set the and-cost to be greater than the or-cost. This parameter setting is also overall better than the first parameter settings (work-guided strategy (1)), because we take the amount of or-work to be done from the or-scheduler as the number of *alternatives* still left to execute. The first parameter setting uses number of live nodes, which is a coarser grained measure and does not give very precise information.

Some of the applications show that there is a limited number of workers at which the speedups are higher for some computations. For example, `bqu8` and `chat` present very good speedups for 10 workers, while the best average speedup for pan2 is produced with 15 workers. This is directly related to the nature of the computation. There are computations that achieve a maximum speedup at a given number of processors. If more processors are added, they will be idling, because the parallelism existing in the benchmark is not enough to justify adding more processors to the execution.

In general, if we use a greater number of processors, the speedups with the reconfigurer drop slightly below the target performance. This is caused mainly for two reasons: lack of parallelism achievable by a larger number of processors; and too many unnecessary reconfigurations due to fine grained work that can delay the execution as we increase the number of processors.

Appendix C

Work-Guided Strategy: Detailed Analysis of Parameters for 10 Processors

In this appendix we show results for the whole benchmark set produced by the work-guided strategy, and analyse the performance of the system for 10 workers by varying some of the parameters described in chapter 7. In section C.1 we discuss how the variation of the and/or threshold factor can affect the performance of the programs. The following section shows how some of the applications have the performance affected by changing the priority of giving work. Section C.4 gives a brief discussion on how "good" or "bad" the reconfigurations are. In section C.5 we analyse more closely some of the applications used as benchmarks. In the next section we give an overview of the variation in execution times between runs of the same query of the same program, under the same reconfigurer strategy and compare these variations with the fixed configuration benchmark. In section C.6 we give a summary of the overall best work-guided strategy with respect to the parameters and/or threshold and or/and threshold (G_a and G_o), and-cost (C_a) and or-cost (C_o), and priority to give work. We analyse the execution times with 1 and 10 processors and compare the speedups produced by the reconfigurer with the and-parallel only and or-parallel only speedups. We discuss the speedups achieved by the benchmarks with 10 processors showing that some of the benchmarks have different degrees of both and- and or-parallelism that can be detected by Andorra-I with the reconfigurer. We also compare the Andorra-I performance with the reconfigurer against the performance obtained by the best fixed configuration of 10 workers. In the end of this section we give an overall performance of the work-guided strategy.

C.1 Changing Parameters

In this section we analyse how the variation of the and/or threshold factor can affect the performance of the system. As this is part of our first experiments before collecting overheads of the system, we fixed the values of C_a in to 10 and C_o to 8. The results shown in this section were collected with the version of the system that uses number of choicepoints as an estimate of or-work. Also we consider the parameters D, i.e., the parallel depth of each goal, and O, the size of each branch, to be 1. This experiment also shows results with the same values for both and/or and or/and threshold.

Table C.1 shows the results for this experiment. If we compare the results shown in table C.1 we can observe that there is a significant variation in speedups when we vary the and/or threshold factor. This means that the system is really exploiting **simultaneously** and- and or-parallelism. As it was mentioned before, the and/or threshold factor is only used when the scheduler detects that there are both kinds of work available to be executed at the same time. The results in table C.1 show that in general for programs that contain more or-parallelism the best value of and/or threshold is 0. For programs that contain more and-parallelism the best value of and/or threshold is 50. The value 20 seems to be the worst for this benchmark set, causing the speedups to be worse for 5 programs. Overall, the value 0 seems to be the best, given that only three programs produced worse speedups than using $G_a = G_o = 3$. A value of and/or threshold equals 0 means that workers will be redeployed to one kind of work or another depending only on if the amount of either one kind of work is greater than zero. This means that we force and- and or-work to be non inter-related.

We can observe that some of the programs that have more or-parallelism can benefit with a change of the and/or threshold parameter `G`. This is the case, for example of the program `ndetdet` that has a non-determinate phase in the beginning of the computation and a determinate phase in the end of each branch of the search tree. Table C.2 shows the amounts of work available during the reconfigurations, when using an and/or threshold value of 3. Table C.3 shows the same information but when using and/or threshold value of 0.

The figures were taken from the best execution of `ndetdet` with different values for the and/or threshold. The benchmark program `ndetdet` running with and/or threshold value of 0 produces best execution time of 4.5 seconds while and/or threshold value of 3 produces best execution time of 5.2 seconds, at 10 processors. The table shows number of reconfigurations (14 for G = 3 and 15 for G = 0) for the two runs, the worker redeployment (column `Rdpment`), the amount of and-work (`And`) available during the reconfiguration and immediately after, and the amount of or-work (`Or`) available during the reconfiguration

	0	3	10	20	50	100	1000
ndetdet	7.811	7.037	6.642	5.947	6.510	6.339	6.242
mixed	3.163	3.053	3.237	3.132	2.820	3.115	3.302
detndet	3.474	3.306	3.041	3.071	3.171	2.946	3.183
bqu8	7.121	4.725	4.683	4.644	4.075	4.363	4.389
chat	6.602	6.583	6.589	6.582	7.090	6.800	6.626
cypher	5.937	5.500	4.925	5.242	5.648	4.695	5.185
mutest	5.989	5.858	5.950	5.534	6.023	6.021	5.896
bqu6	2.929	2.360	2.373	2.508	2.572	2.848	1.673
flypan2	4.050	6.069	5.801	3.529	5.741	5.669	5.874
flypan4	4.847	4.995	4.604	4.771	4.964	3.974	4.528
scanner	5.037	4.787	4.565	4.933	4.996	4.317	4.589
road_markings	3.821	3.805	4.034	3.960	4.171	3.810	3.702
bcnet	4.089	3.853	3.934	3.833	3.821	3.868	3.915
cross11	2.867	2.917	2.782	2.784	3.281	2.861	2.989
flypan5	3.232	3.073	3.325	3.282	3.285	3.258	3.254
cross6	2.121	2.069	2.067	1.891	2.286	2.285	1.799
Average	2.070	3.344	2.264	2.255	2.172	2.335	2.284

Table C.1: SPEEDUPS WHEN VARYING AND/OR THRESHOLD

and immediately after. As we can observe the quality of reconfigurations is much better when we use and/or threshold value of 0. As this benchmark has more or-parallelism, the workers should become masters in the beginning and eventually become slaves later on in the execution as the number of alternatives start to decrease. Table C.3 shows exactly this phenomenon. What happens for these two executions is that with $G = 3$, slaves do not get as many chances to become a master as they should, because the test $or \geq 3 * and$ is more restrictive than $or \geq 0$.

C.2 Varying Priority to Give Work

In this section we discuss how different priorities to choose between and- and or-work can affect the system's performance. The algorithm 7.1 used so far to collect results is giving priority to and-work when slaves call the reconfigurer after running out of and-work in their teams. A consequence of this is that, if there is some or-work available and this is

Rcfg	Wkr	Rdpment	And	Or
0	7	S→M	0,0	1,1
1	4	S→M	0,0	2,3
2	0	M→S	7,10	0,10
3	5	S→S	6,7	0,9
4	2	S→S	4,14	0,10
5	8	S→S	4,13	0,10
6	6	S→S	6,6	0,10
7	0	S→S	7,8	0,10
8	8	S→S	7,12	0,9
9	2	S→S	7,12	0,9
10	1	M→S	2,9	5,5
11	1	S→S	2,6	0,5
12	6	S→S	1,4	0,5
13	3	S→S	1,2	0,0

Table C.2: AMOUNT OF WORK AVAILABLE DURING RECONFIG, PGM `ndetdet`, G = 3

much bigger than the current amount of and-work available, the slave would be held in its original team until the determinate phase finishes, without the slave knowing if there was good amount of or-work available.

We made three different experiments by slightly modifying the main algorithm to change slaves to give different levels of priority to or-work and give no priority to either kind of work. The results are shown in Table C.4.

Giving different levels of priority to Or-work With this experiment we wanted to show how the system is sensitive to changes in priorities to give work. Columns `Vh_orpri` and `Avg_orpri` in table C.4 shows the results for this experiment. `Vh_orpri` means that the first thing a slave tries to do when it runs out of work is to become a master and create a new team, independent on the amounts of work available. `Avg_orpri` means that the first thing a slave tries to do when it runs out of work is to become a master and create a new team, but depending on the amounts of work available.

If we observe table C.4, we notice that the 5 programs in subgroup 2, that contain mainly or-parallelism, do not produce very good speedups compared with the best fixed configu-

Rcfg	Wkr	Rdpment	And	Or
0	1	S→M	0,0	1,2
1	8	S→M	0,0	1,2
2	3	S→M	0,3	2,4
3	6	S→M	0,3	2,5
4	0	S→M	0,4	3,5
5	2	S→M	44,47	16,15
6	9	S→M	44,45	16,15
7	9	M→S	8,54	17,17
8	9	S→S	8,33	0,0
9	1	M→S	6,34	14,14
10	7	M→S	2,5	10,10
11	8	M→S	2,3	0,0
12	0	M→S	1,4	0,0
13	6	M→S	1,2	0,0
14	3	M→S	2,2	0,0

Table C.3: AMOUNT OF WORK AVAILABLE DURING RECONFIG, PGM `ndetdet`, G = 0

ration (table C.5. By giving priority to or-work the execution improved in average 12.46% if we take the average of the speedups for the programs that were not performing well with the strategy that gives priority to and-work. Therefore the degradation that was in average 17.9% before compared with the best fixed configuration drops to around 5%, what is quite reasonable. The other programs, that contain mainly and-parallelism, were also tested in order to verify if this change in the algorithm would affect their performance. The speedups dropped for two programs when not giving priority to and-work, but increased for the other remaining six programs. Therefore, except for programs `flypan2` and `flypan4`, we can conclude that in general it is better to give priority to or-work than to and-work.

This conclusion might no be correct if Andorra-I allowed slaves to go out during the determinate phase. One of the main reasons why the strategy giving priority to or-work performs better is that by giving priority to and-work, once a slave commits to be a slave in another team, it can only be redeployed when the determinate phase finishes. If meanwhile there was some or-parallel work available the slave does not have a chance to get it.

	And_pri	Vh_orpri	Avg_orpri	No_pri
ndetdet	*7.037*	*4.942*	*6.862*	*6.310*
mixed	*3.053*	*2.281*	*3.153*	*3.430*
detndet	*3.306*	*3.525*	*3.374*	*2.686*
bqu8	*4.725*	*6.842*	*5.271*	*2.007*
chat	*6.583*	*7.218*	*6.714*	*6.882*
cypher	*5.500*	*5.830*	*5.578*	*4.083*
mutest	*5.858*	*6.331*	*5.934*	*6.012*
bqu6	*2.360*	*2.897*	*2.779*	*2.160*
flypan2	*6.069*	*5.885*	*5.645*	*4.112*
flypan4	*4.995*	*4.853*	*4.473*	*1.889*
scanner	*4.787*	*5.461*	*5.093*	*1.928*
road_markings	*3.805*	*4.275*	*4.190*	*4.076*
bcnet	*3.853*	*4.038*	*3.930*	*1.747*
cross11	*2.917*	*3.040*	*2.875*	*1.643*
flypan5	*3.073*	*3.476*	*3.455*	*1.781*
cross6	*2.069*	*2.134*	*2.070*	*1.659*
Average	*3.344*	*5.164*	*5.077*	*5.164*

Table C.4: SPEEDUPS WHEN VARYING PRIORITY TO GIVE WORK

Do not Give Priority to either kind of work Column `No_pri` in table C.4 shows how the speedups are affected if we use an algorithm that makes global decisions without giving priority to either kind of work (the algorithm was shown in chapter 7, figure 7.6). By using this algorithm we can see that the speedups drop to almost all the programs. This is explained by the fact that the decision algorithm is more general in order to decide between and- and or-work. The previous algorithms do decisions locally and consequently incur less overhead.

C.3 Using a better Estimate for the amount of or-work

In this section we show results for 10 processors when using a better estimate of size of or-work at runtime. We changed the or-scheduler to give to the reconfigurer the number of alternatives left on the tree, instead of number of live nodes.

Pgm	9S	2M8S	3M7S	4M6S	5M5S	6M4S	7M3S	8M2S	9M1S	10M
ndetdet	2.037	4.673	5.913	5.812	7.971	7.825	8.272	8.159	7.717	8.487
mixed	0.980	1.920	2.642	3.574	3.819	3.997	3.587	3.616	3.676	3.584
detndet	1.044	1.622	2.052	2.187	2.387	2.528	2.905	3.012	1.903	2.032
bqu8	1.527	2.858	3.937	4.802	5.487	5.832	6.165	6.575	7.216	7.516
chat	0.666	1.578	2.502	3.204	3.866	4.564	5.451	6.108	6.539	7.334
cypher	1.147	2.164	2.875	3.503	4.046	4.492	4.942	5.381	5.729	5.922
mutest	0.584	1.386	2.071	2.718	3.450	4.180	4.726	5.382	5.712	5.959
bqu6	1.499	2.562	2.940	2.950	3.132	3.383	3.352	3.603	3.950	3.137
flypan2	6.203	4.105	3.507	1.955	1.964	1.930	0.984	1.814	0.999	1.007
bt_cluster	6.002	4.353	3.947	3.916	1.995	1.994	2.017	1.992	1.973	1.011
flypan4	5.079	3.725	3.305	1.973	1.947	1.995	1.064	1.965	1.063	1.062
scanner	5.009	5.460	4.992	5.118	4.087	3.962	3.835	3.922	4.142	3.226
road_markings	4.887	3.479	2.484	3.015	1.790	1.772	1.761	1.766	1.684	0.965
bcnet	3.226	3.763	3.336	2.852	2.767	2.007	2.066	2.452	2.376	2.033
cross11	2.666	1.647	1.615	1.675	1.669	1.639	1.589	1.609	1.402	1.112
flypan5	2.629	2.878	2.506	2.041	2.139	2.162	1.349	1.356	1.374	1.335
cross6	2.529	2.260	2.095	1.473	1.342	1.117	1.457	1.392	0.967	0.964
H.mean	1.656	2.505	2.787	2.682	2.574	2.528	2.230	2.590	2.125	1.836

Table C.5: SPEEDUPS FOR FIXED CONFIGURATIONS (10 WORKERS)

Table C.6 shows the results comparing the work-guided strategy that uses number of choicepoints as an estimate for amount of or-work (`WBSch`), with the work-guided strategy that uses number of alternatives as an estimate for the amount of or-work (`WBSalt`). In general, we can observe that utilising a better estimate for the size of or-work affects positively the performance with only two results being worse than the work-guided strategy that uses choicepoints as estimate for amount of or-parallelism. However, some of the results are still below the speedups produced by the best fixed configuration.

C.4 Study of the Reconfigurations

In this section we evaluate the quality of the reconfigurer by investigating how "good" or "bad" a reconfiguration is. The results were taken from the best of the runs with 10 workers for all the applications using the work-guided strategy that gives priority to and-

	WBSch	WBSalt	WBSalt/WBSch
ndetdet	7.037	7.214	1.025
mixed	3.053	4.004	1.312
detndet	3.306	3.793	1.147
bqu8	4.725	5.063	1.072
chat	6.583	6.326	0.961
cypher	5.500	6.116	1.112
mutest	5.858	6.025	1.029
bqu6	2.360	2.920	1.237
flypan2	6.069	6.307	1.039
flypan4	4.995	4.961	0.993
scanner	4.787	5.414	1.131
road_markings	3.805	4.269	1.122
bcnet	3.853	3.873	1.005
cross11	2.917	3.127	1.072
flypan5	3.073	3.382	1.101
cross6	2.069	2.260	1.092

Table C.6: USING NUMBER OF ALTERNATIVES AS AN ESTIMATE OF OR-WORK

work, uses choicepoints as an estimate for or-work and $G_a = G_o = 3$. A reconfiguration is annotated whenever the reconfigurer decides to change a worker (change a slave to master, a master to slave or redeploy slaves between teams). We will name a "good" reconfiguration the one whose amount of work (and- or or-) is at least greater than C_a or C_o before and after the decision. Likewise we will call a "bad" reconfiguration the one whose size of work after the decision is less than it used to be before the decision.

Table C.7 shows for each application, the profile of reconfigurations for the best run. Columns $M \rightarrow S$, $S \rightarrow M$ and $S \rightarrow S$ show, respectively, the number of actual reconfigurations from master to slave, from slave to master and changes between teams. The columns `good` and `bad` show the number of good and bad reconfigurations for the best run of each application. Column `% good of total` shows the percentage of good reconfigurations for each application. The last 2 columns show the actual number of calls to the reconfigurer, made from slaves and from masters, to change their original configuration.

For the programs that have more or-parallelism (first and second group of applications) the percentage of good reconfigurations is above 50%. This means that more than half of the reconfigurations were "good". The exceptions are for `chat` and `mutest`. The program `chat` had a poorer speedup with the reconfigurer than with the fixed teams configuration, and so did `mutest`. As `mutest` did only one reconfiguration, its speedup was not so much affected as `chat`'s speedup.

The and-parallel applications, with exception of `cross6` and `cross11`, all had very bad behaviour with respect to reconfigurations, but even so produced good speedups results.

	Reconfigurations					%good	Calls to Reconfigure	
	M→S	S→M	S→S	good	bad	of total	from sl	from ma
ndetdet	2	2	10	10	4	71.43	37	771
mixed	2	1	6	5	4	55.55	33	2457
detndet	6	9	7	19	3	86.36	48	804
bqu8	7	8	375	339	36	86.92	804	551
chat	2	2	0	0	4	00.00	4	1389
cypher	11	8	51	44	26	62.85	168	1614
mutest	1	0	0	0	1	00.00	1	1634
bqu6	4	4	74	48	34	58.53	161	746
flypan2	2	2	24	2	26	7.14	102	189
flypan4	9	9	201	41	178	18.72	496	592
scanner	5	5	303	26	287	8.30	782	2148
road_markings	11	10	32	14	39	26.41	103	5197
bcnet	6	5	147	51	107	32.28	570	12137
cross11	6	6	14	17	9	65.38	89	3259
flypan5	67	68	797	193	739	20.71	2822	9993
cross6	1	4	8	9	4	56.25	50	398

Table C.7: NUMBER OF GOOD AND BAD RECONFIGURATIONS

This classification of 'bad' and 'good' reconfigurations is only to illustrate how the reconfigurer is reacting to the system. In fact good or bad reconfigurations depend on the future amount of computation, which is not available.

C.5 Case Studies

Now let us examine some applications more closely. We chose applications that have good or-parallelism and good and-parallelism that run well and not so well with dynamic reconfiguration (detndet and ndetdet).

Table C.8 shows execution times, in seconds, for all different combinations of workers for application `detndet` demonstrating that no fixed configuration can achieve the best time achieved by the reconfigurer. The average time to run `detndet` with the best fixed configuration (8 masters and 2 slaves) is 3.859 seconds against 3.502 seconds with dynamic reconfiguration (in average 9.25% better). The best times for these runs (fixed and dynamic) give a difference of about 23% in favour of the reconfigurer. We will see later that this time can be even reduced by decreasing the time a processor spends idle in the system. This program has an execution tree structure similar to the example shown in figure 6.1 and discussed in section 6. As there are two distinct phases, determinate followed by non-determinate phase, any fixed configuration would not do well, because the 2 phases require different arrangement of workers. Therefore this program is favoured by a dynamic reconfiguration of workers.

9S	2M8S	3M7S	4M6S	5M5S	6M4S	7M3S	8M2S	9M1S	10M
11.093	7.139	5.641	5.348	4.849	4.600	3.992	**3.859**	6.093	5.707

Table C.8: EXECUTION TIMES (SECONDS) FOR `detndet`, FIXED

Figure C.1 shows that the workers rearranged from one phase to the other to produce a better speedup than the fixed configuration. The picture was produced with the VisAndOr tool developed in the Polytechnic University of Madrid by Luis Gómez [51].

The other benchmark program studied closer is the program `ndetdet`. This program produced the best average speedup with the reconfigurer when using the value zero for G_a and G_o (the and/or and or/and thresholds). Table C.9 shows the total reductions performed by each processor for the best run of this program with threshold values of 0 and 3. When the and/or threshold value is 0, we can see that the number of reductions performed per each worker is about the same, while when we use an and/or threshold value of 3, the total number of reductions is not so well distributed among the workers. We also show the total number of reconfigurations with percentages of idle, working

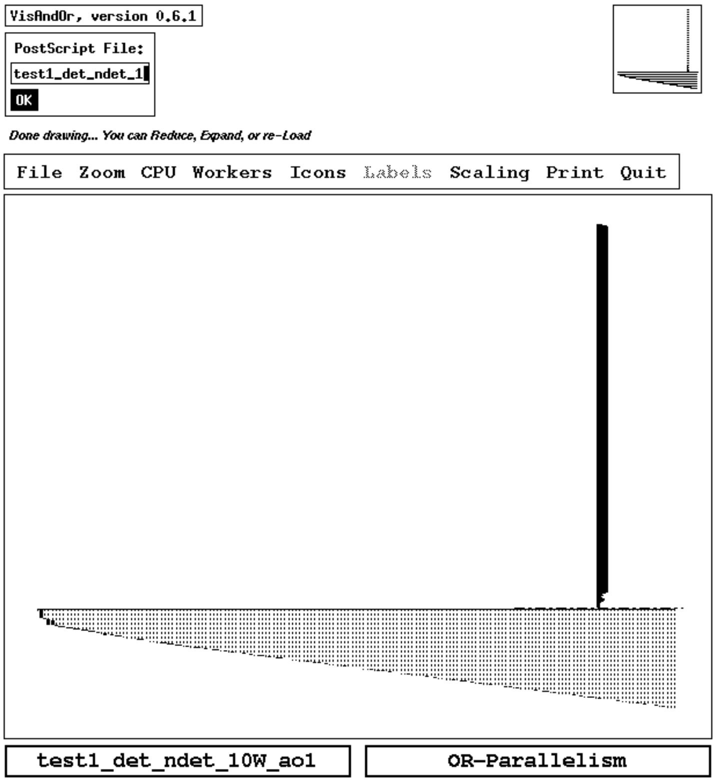


Figure C.1: `detndet` VISANDOR EXECUTION TREE and finding and-work times. With and/or threshold value of zero, the processors spend 91.50% of their total execution time performing reductions, i.e., doing useful work, while with and/or threshold value of 3, the processors spend more time idle.

Pgm	G	Total Reductions per Worker									
		0	1	2	3	4	5	6	7	8	9
ndetdet	3	17427	14294	20473	20897	17222	16975	14699	14949	17004	17176
ndetdet	0	17618	16474	16937	17526	17112	16886	17584	17752	17633	15594

Pgm	G	Reconfigurations			Calls for Changes		%work	%idle	%find_and
		M→S	S→M	S→S	from S	from M			
ndetdet	3	2	2	10	37	771	84.86	9.25	3.51
ndetdet	0	7	7	1	22	1428	91.50	5.09	1.44

Table C.9: USING DIFFERENT PARAMETERS FOR `ndetdet`

The benchmark program `ndetdet` running with and/or threshold value of 3 produced more reconfigurations than `ndetdet` running with and/or threshold value of 0. Although the percentage of good reconfigurations is quite high for both runs (12 good out of 15 for G equals 0, and 10 out of 14 for G equals 3), we need also to consider the time taken between two subsequent reconfigurations. This is not considered when counting good or bad reconfigurations.

The best run for this program is with the fixed configuration version with 10 masters. This is explained because this program creates a choicepoint in the beginning that has 10 alternatives. If each worker takes one of the branches, we would expect to have the best execution with a fixed configuration of 10 masters. Unfortunately the reconfigurer does not recognise this structure of the tree and workers rearrange as masters too late to be able to start the left branches sooner and finish the execution in a better time. If we give priority to or-work the execution improves, but still does not reach the best fixed configuration.

C.6 Summary of the Work-Guided Strategy

Table C.10 shows the speedups obtained by Andorra-I when using fixed teams and the reconfigurer. All results are relative to the speed of the uniprocessor version. Column `10M` corresponds to Andorra-I running with a fixed configuration of `10` masters while `9S` corresponds to Andorra-I running with a fixed configuration of 1 master and `9` slaves. These data show how much or-parallelism only and and-parallelism only exist, respectively, in each application, for 10 workers. The data related to the reconfigurer (`10W`) shown in the tables were obtained with all workers initially configured as only one team. Therefore, `10W` means that all programs started with a configuration of 1 master and 9 slaves. If an application creates a choicepoint on starting the query, the processors will be reconfigured. The results presented in the table are the best for each application, with different parameters and priority to give work and different costs. Each benchmark had a best average speedup depending on the values of the parameters discussed before. `G` represents the parameters G_a and G_o (the and/or threshold and or/and threshold). `st` indicates the estimate used to get the amount of or-work from the or-scheduler ('alt' gets the number of alternatives from the or-scheduler while 'ch' gets the number of choicepoints). The parameters C_a and C_o were described before. `D` is the parallel depth estimated for each goal. `O` is the grain size of each alternative. And `c` is the grain size for each choicepoint, when we use number of choicepoints as an estimate for or-work. The symbol 'vh' is used to indicate that the version that achieved the best speedup gives very high priority to or-work.

The first column of the table shows the classes of applications. The first class (artificial programs) consists of well behaved artificial examples used to study the strategy of the reconfigurer. From this group, program `detndet` demonstrated to produce a tree that is not possible to be executed optimally with fixed teams configuration. In contrast Andorra-I with the reconfigurer produced a poor performance result to `ndetdet` and `bqu8`.

The other group contains programs that are more realistic and is divided into two subgroups according to the degree of and- and or-parallelism available. The first subgroup contains more or-parallelism. The second subgroup contains more and-parallelism (the last programs of this second group have little or no parallelism at all). From these two subgroups, Andorra-I running with the reconfigurer achieved comparable performance with the best fixed team configuration, regardless of the amount of and- or or-parallelism existing in the application. The worst degradation was 25.8% for `bqu6`. For the subgroup of programs that present mainly or-parallelism we conclude that there is not much initial overhead of arranging the workers in 10 teams, given that we started with just one team of 10 workers. In general, programs that contain very little parallelism (low granularity), e.g, `bqu6` and `cross6` produce better results for a fixed configuration of workers. This is obviously explained by the fact that for small programs, the reconfigurer introduces an extra overhead that is noticeable for very small programs.

For some of the programs that contain mainly or-parallelism, the speedups for the work-guided strategy are quite good, but not so good as the results produced by the fixed team

	10M	9S	Target		10W	Parameters	R
						G-st-C_a-C_o-D-O-c	
ndetdet	8.487	2.037	8.487	*(10M)*	9.831	0-alt-10-8-1-1-1	1.158
mixed	3.584	0.980	3.997	*(6M4S)*	4.004	3-alt-10-8-1-1	0.998
detndet	2.032	1.044	3.012	*(8M2S)*	3.793	3-alt-10-8-1-1	0.794
bqu8	7.516	1.527	7.516	*(10M)*	7.121	0-ch-10-8-1-1-1	1.055
chat	7.334	0.666	7.334	*(10M)*	7.218	3-ch-10-8-1-1-1,vh	1.016
cypher	5.922	1.147	5.922	*(10M)*	6.116	3-alt-10-8-1-1	0.997
mutest	5.959	0.584	5.959	*(10M)*	6.025	3-alt-10-8-1-1	0.990
bqu6	3.137	1.499	3.950	*(9M1S)*	2.929	0-ch-10-8-1-1-1	1.349
flypan2	1.007	6.203	6.203	*(9S)*	6.307	0-alt-3-7-10-5	0.984
flypan4	1.062	5.079	5.079	*(9S)*	4.995	3-alt-10-8-1-1	1.017
scanner	3.226	5.009	5.485	*(2M8S)*	5.650	0-alt-3-7-10-5	0.971
road_markings	0.965	4.887	4.887	*(9S)*	4.320	0-alt-3-7-10-5	1.131
bcnet	2.033	3.226	3.746	*(2M8S)*	4.089	0-ch-10-8-1-1-1	0.916
cross11	1.112	2.666	2.666	*(9S)*	3.281	50-ch-10-8-1-1-1	0.813
flypan5	1.335	2.629	2.862	*(2M8S)*	3.476	50-ch-10-8-1-1-1	0.823
cross6	0.964	2.529	2.529	*(9S)*	2.286	50-ch-10-8-1-1-1	1.106
H.mean	1.934	1.584	4.335	—	4.436		

Table C.10: SPEEDUPS – OR-ONLY, AND-ONLY, RECONFIGURER, BEST FIXED (10 WORKERS)

configuration. This is due to the fact, already mentioned, that the reconfigurer sometimes turns masters into slaves, because it gets mislead by the fine grained and-parallelism available in these programs.

The reconfigurer produces better results for 9 out of 16 benchmark programs showing that there is no fixed configuration capable of exploiting the parallelism available in those applications. The other programs produced results compared with the best fixed configuration. The program `scanner`, that contains a reasonable amount of and- and or-parallelism, did not produce very good speedups, as it might have been expected given that this program has a good amount of and-parallelism and or-parallelism (see columns `9S` and `10M` in table C.10). However the result shown is improved when we change the and/or threshold value to 0 and give very high priority to or-work (see sections C.1 and C.2). We did not try a combination of both runs, but we believe the combination would result in an even better performance.

The line named “H.mean” represents the average speedups and show that overall the results produced by Andorra-I under the reconfigurer strategy are better than those obtained with a fixed configuration of teams adjusted to exploit and-parallelism only and or-parallelism only. The results are also comparable with the best execution time produced by the best fixed configuration, with an average absolute degradation of 0.28%.

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 reconfigurer 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 is not capable of exploiting the kind of parallelism available in the application (e.g., this benchmark is not part of this thesis, but the worst time to execute naïve reverse in Andorra-I with fixed teams configuration is achieved when we use 10 masters, because this is a totally determinate program). 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. As we can see from Table C.10 there is no single fixed configuration that fits all applications, and yet produces the best speedups. This is obviously concluded from the fact that programs that have and-parallelism predominantly need a configuration with more slaves. Programs that have or-parallelism predominantly need a configuration with more masters. It is impossible for these cases to use a common fixed configuration that works well for both kinds of programs.

The applications `flypan` and `bqu` are interesting to be analysed. We tried three different queries for the same program `flypan` and each one produces different results with best speedups being achieved by different fixed teams configuration. `flypan4` and `flypan2` produces best speedups with a fixed configuration of `9` slaves, while `flypan5` produces the best speedup with fixed configuration of `2` masters and `8` slaves. By using the reconfigurer starting from one single worker configuration for all queries we achieve similar results. The reconfigurer dynamically reconfigures the workers and produces similar speedups for `flypan4`, and better speedup for the queries `flypan2` and `flypan5`. The same happens to the `bqu` program. The same program running with two different queries (two different board sizes) requires two different fixed configurations to reach the best speedup (`bqu6` requires a fixed configuration of 9 masters and 1 slave while `bqu8` requires a fixed configuration of 10 masters). The reconfigurer although not performing very well for these two queries compared with the best fixed configuration, still achieves reasonable speedups starting from a single team configuration and dynamically changing it along the execution.

Not surprisingly, most of the best results shown in table C.10 are achieved with $C_a = 10$ and $C_o = 8$, although the best values indicated by the performance analysis of Andorra-I are around 3 and 7, respectively. This phenomenon is explained by the fact that Andorra-I has the limitation of not allowing slaves to be redeployed while they are in a determinate phase. Therefore a value of $C_a = 10$ is more restrictive than a value of $C_a = 3$, which means that slaves will be redeployed less often from one team to another, and have more chances to become masters. In other words, having a value 10 for C_a means giving lower priority to and-work.

Regarding the parameters G_a and G_o, our conclusion is that workers should be redeployed to any kind of work available since they are idle. Therefore a value of G_a and G_o equals zero produces better results for most of the applications, as workers do not care about the quantities of and- and or-work related to each other. The work-guided strategy also proved to behave better for an algorithm that gives priority to or-work. This works well probably for the same reason that giving higher value to C_a makes the system work better. Another reason why giving priority to or-work produces better results is that usually or-work has bigger granularity than and-work. In this case, the worker will spend more time working as a master in a branch than competing with other workers in the same team to find and-work.

Appendix D

Source of some Benchmark Programs

This appendix contains the source files with respective queries for some of the benchmarks used in this thesis. More benchmarks are available via ftp from University of Bristol. Please send a mail to *rong@acrc.bris.ac.uk* or *beaumont@acrc.bris.ac.uk* if you wish a copy of other benchmarks.

```
/******************************/
/* ndetdet                    */
/* query: ?- main1.           */
/******************************/

sics :- cputime(T1),
        main,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

main :- p(M,N),
        fail.
main.

main1 :- p(M,N),fail.
main1.

p(X,Y) :- q(X,Z), r(Z,Y).

q(X,Z) :- s1(X,Z).
q(X,Z) :- s2(X,Z).
q(X,Z) :- s3(X,Z).
q(X,Z) :- s4(X,Z).
q(X,Z) :- s5(X,Z).
q(X,Z) :- s6(X,Z).
q(X,Z) :- s7(X,Z).
q(X,Z) :- s8(X,Z).
q(X,Z) :- s9(X,Z).
q(X,Z) :- s0(X,Z).

r(b,Z) :- t1(b,Z).
r(c,Z) :- t2(c,Z).
```

```
r(d,Z) :- t3(d,Z).
r(e,Z) :- t4(e,Z).
r(f,Z) :- t5(f,Z).
r(g,Z) :- t6(g,Z).
r(h,Z) :- t7(h,Z).
r(i,Z) :- t8(i,Z).
r(j,Z) :- t9(j,Z).
r(k,Z) :- t0(k,Z).

s1(a,b).
s1(a,b1).
s1(a,b2).
s1(a,b3).

s2(b,c).
s2(a,c1).
s2(a,c2).
s2(a,c3).

s3(c,d).
s3(a,d1).
s3(a,d2).
s3(a,d3).

s4(d,e).
s4(a,e1).
s4(a,e2).
s4(a,e3).

s5(e,f).
s5(a,f1).
s5(a,f2).
s5(a,f3).

s6(f,g).
s6(a,g1).
s6(a,g2).
s6(a,g3).

s7(g,h).
s7(a,h1).
s7(a,h2).
s7(a,h3).

s8(h,i).
s8(a,i1).
s8(a,i2).
s8(a,i3).

s9(i,j).
s9(a,j1).
s9(a,j2).
s9(a,j3).

s0(j,k).
s0(a,k1).
s0(a,pk).
s0(a,k3).

t1(b,c) :- loop(1000), loop(2000), loop(1500), loop(1200).
t1(c,c).

t2(c,d) :- loop(1000), loop(2000), loop(1500), loop(1200).
t2(d,d).

t3(d,e) :- loop(1000), loop(2000), loop(1500), loop(1200).
t3(e,e).
```

```
t4(e,f) :- loop(1000), loop(2000), loop(1500), loop(1200).
t4(f,f).

t5(f,g) :- loop(1000), loop(2000), loop(1500), loop(1200).
t5(g,g).

t6(g,h) :- loop(1000), loop(2000), loop(1500), loop(1200).
t6(h,h).

t7(h,i) :- loop(1000), loop(2000), loop(1500), loop(1200).
t7(i,i).

t8(i,j) :- loop(1000), loop(2000), loop(1500), loop(1200).
t8(j,j).

t9(j,k) :- loop(1000), loop(2000), loop(1500), loop(1200).
t9(k,k).

t0(k,l) :- loop(1000), loop(2000), loop(1500), loop(1200).
t0(l,l).

loop(0).
loop(N) :- N > 0,
           New_N is N - 1,
           loop(New_N).

/* ?- p(X,Y). */
```

```
/************************************/
/* mixed                            */
/* query: ?- main2.                 */
/************************************/

sics :- cputime(T1),
        main,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

main:-  p(M,N),
        fail.
main.

main1 :- p(M,N),fail.
main1.

p(X,Y) :- loop(100),loop(200),loop(500),q(X,Y).
p(X,Y) :- f(X,Z),r(Z,Y).
p(e,f) :- loop(500).
p(X,Y) :- m(Z),loop(Z,100),loop(Z,200),loop(500),v(X,Y).

q(a,b) :- loop(500).
q(X,Y) :- s(X,Y).

s(a1,b1) :- loop(500).
s(a2,b2) :- loop(100),loop(500),loop(200).

t(c1,d1) :- loop(100).
t(X,Y) :- u(X,Y).
t(c2,d2) :- loop(100),loop(200),loop(500).

u(X,d) :- loop(500),loop(500),loop(500).

v(g,h) :- loop(500).
v(g1,h1) :- loop(500).

f(c,c1).

m(b).

r(c1,Y) :- t(X,Y).
r(d,d1).


loop(0).
loop(N) :- N > 0,
           New_N is N - 1,
           loop(New_N).

loop(b,0).
loop(b,N) :- N > 0,
           New_N is N - 1,
           loop(b,New_N).

loop(c,0).
loop(c,N) :- N > 0,
           New_N is N - 1,
           loop(c,New_N).

/* ?- p(X,Y). */
```

```
/**********************/
/* detndet            */
/* query: ?- main.    */
/**********************/

sics :- cputime(T1),
        main,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

main :- p(M,N),
        fail.
main.

main1 :- p(M,N),fail.
main1.

p(X,Y) :- s(X,Z), r(Z,Y).

/*q(X,Y) :- s(X,Y).*/

s(a,b) :- loop(2000), loop(500), loop(1500),
          loop(10), loop(100), loop(50), loop(500),
          loop(2000), loop(30), loop(90).

loop(0).
loop(N) :- N > 0,
           New_N is N - 1,
           loop(New_N).

r(b,c) :- loop(100).
r(a,b) :- loop(100).
r(a,c) :- loop(100).
r(a,d) :- loop(100).
r(b,e) :- loop(100).
r(b,f) :- loop(100).
r(b,c) :- loop(100).
r(a,b) :- loop(100).
r(a,c) :- loop(100).
r(a,d) :- loop(100).
r(b,e) :- loop(100).
r(b,f) :- loop(100).
r(b,c) :- loop(100).
r(a,b) :- loop(100).
r(a,c) :- loop(100).
r(a,d) :- loop(100).
r(b,e) :- loop(100).
r(b,f) :- loop(100).
r(b,c) :- loop(100).
r(a,b) :- loop(100).
r(a,c) :- loop(100).
r(a,d) :- loop(100).
r(b,e) :- loop(100).
r(b,f) :- loop(100).
r(b,c) :- loop(100).
r(a,b) :- loop(100).
r(a,c) :- loop(100).
r(a,d) :- loop(100).
r(b,e) :- loop(100).
r(b,f) :- loop(100).
r(b,c) :- loop(100).
r(a,b) :- loop(100).
r(a,c) :- loop(100).
r(a,d) :- loop(100).
r(b,e) :- loop(100).
```

```
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
```

```
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
r(a,b)  :- loop(100).
r(a,c)  :- loop(100).
r(a,d)  :- loop(100).
r(b,e)  :- loop(100).
r(b,f)  :- loop(100).
r(b,c)  :- loop(100).
```

```
r(a,b) :- loop(100).
r(a,c) :- loop(100).
r(a,d) :- loop(100).
r(b,e) :- loop(100).
r(b,f) :- loop(100).
r(b,c) :- loop(100).
r(a,b) :- loop(100).
r(a,c) :- loop(100).
r(a,d) :- loop(100).
r(b,e) :- loop(100).
r(b,f) :- loop(100).

/* ?- p(X,Y) */




/*********************/
/* bqu               */
/* queries: ?- main1.*/
/*          ?- main2.*/
/*********************/

sics1:- cputime(T1),
        main1,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

main1 :-queens(6,M),
        fail.
main1.

sics2:- cputime(T1),
        main2,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

main2:- queens(8,M),
        fail.
main2.

:- mode queens(+, -).
queens(N, V) :-
  board(0, N, V, L, R, VRingsTop, VRingsEnd),
  to_start(VRingsTop),
  to_end(VRingsEnd).

:- mode to_start(+).
to_start([]).
to_start([H|T]) :-
  H = begin,
  to_start(T).

:- mode to_end(+).
to_end([]).
to_end([H|T]) :-
  H = end,
  to_end(T).

:- mode board(+, +, -, -, -, -, -).
board(K, N, V, LT, [R|RT], VRingsL, VRingsR) :- K < N,
  K1 is K + 1,
  row(K1, 0, N, H, V, [L|LT], [R|RT], begin, end, VRingsL, Temp),
```

```
  board(K1, N, V, [L|LT], RT, Temp, VRingsR).
board(N, N, V, L, R, VRings, VRings).

:- mode row(+, +, +, -, -, -, -, -, -, -, -).
row(I, J, JMax, H, [V|VT], [L|LT], [R|RT], HL, HR,
                           [VL|VRings], [TempV|TempRings]) :- J < JMax,
  J1 is J + 1,
  cell(I, J1, H, V, L, R, HL, TempH, VL, TempV),
  row(I, J1, JMax, H, VT, LT, RT, TempH, HR, VRings, TempRings).
row(I, JMax, JMax, H, [], L, R, HL, HL, [], []).

cell(I, J, J, I, I, I, begin, end, begin, end).
cell(_, _, _, _, _, _, Hc, Hc, Vc, Vc).


/*
:- mode cell(+, +, +, +, +, +, +, +, +, +).
cell(I, J, H, V, L, R, Left1, Right1, Left2, Right2) :- H \== J, !,
  Left1 = Right1, Left2 = Right2.
cell(I, J, H, V, L, R, Left1, Right1, Left2, Right2) :- V \== I, !,
  Left1 = Right1, Left2 = Right2.
cell(I, J, H, V, L, R, Left1, Right1, Left2, Right2) :- L \== I, !,
  Left1 = Right1, Left2 = Right2.
cell(I, J, H, V, L, R, Left1, Right1, Left2, Right2) :- R \== I, !,
  Left1 = Right1, Left2 = Right2.
cell(I, J, H, V, L, R, begin, end, Left, Right) :-
  H = J, V = I, L = I, R = I, Left = begin, Right = end.
cell(I, J, H, V, L, R, Left, Right, begin, end) :-
  H = J, V = I, L = I, R = I, Left = begin, Right = end.
*/
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%    DSSA System - Decoding Simple Substitution ciphers in Andorra-I
%%                            - Rong Yang  (tmp version)
%%    (C) Jan. 89, Dept. Computer Science Bristol University
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*********************/
/* cypher            */
/* query: ?- goal(24)*/
/*********************/

goal(MaximumErr):-
       init(CipherText, ReorderedCT, SubsTable, TopN, FreqList,
                                                   MaximumErr),
        sub_gen(TopN, FreqList),
        look_for_dic(ReorderedCT, ErrMessage),
        consistent_check(SubsTable),
        err_counter(0, MaximumErr, ErrMessage),
       print_out(CipherText, MaximumErr).
goal(X):-  write('Sorry please give me longer ciphertext').

%%    In look_for_dic( Text, ErrMessages ):
%%    Text is an input ciphertext;
%%    ErrMessages is a list which receives err-messages from
%%        look_for processes.

look_for_dic([],[]).
look_for_dic([SH|SL],E):- get_word(W,[],[SH|SL],Rest),
        look_for(W,E1),
        look_for_dic(Rest,E2),
       merge_err(E1,E2,E).
```

```
merge_err(L1,L2,[]) :- /, L1 = [], L2 = [].
merge_err([],L,L1) :- / , L1 = L.
merge_err(L,[],L1) :- / , L1 = L.
merge_err([H|T],L,L1) :- / ,
        L1 = [H|NT],
        merge_err(T,L,NT).
merge_err(L,[H|T],L2) :- / ,
        L2 = [H|NT],
        merge_err(L,T,NT).

get_word(W,W, [], []).
get_word(W,W, [' '|Rest], Rest).
get_word(W,I, [X|T], Rest) :- X \== ' ',
        get_word(W,[X|I], T,Rest).

look_for([],  ErrMess).
look_for([X1], ErrMess):-
        word1(X1,ErrMess).
look_for([X1,X2], ErrMess):-
        word2(X2,X1,ErrMess).
look_for([X1,X2,X3], ErrMess):-
        word3(X3,X2,X1,ErrMess).
look_for([X1,X2,X3,X4], ErrMess):-
        word4(X4,X3,X2,X1,ErrMess).
look_for([X1,X2,X3,X4,X5], ErrMess):-
        word5(X5,X4,X3,X2,X1,ErrMess).
look_for([X1,X2,X3,X4,X5,X6], ErrMess):-
        word6(X6,X5,X4,X3,X2,X1,ErrMess).
look_for([X1,X2,X3,X4,X5,X6,X7], ErrMess):-
        word7(X7,X6,X5,X4,X3,X2,X1,ErrMess).

%% and so on ...

%%     consistent_check(L) makes sure
%%     that no of pair elements in L are same.

consistent_check([]).
consistent_check([X|T]):-
        row_check(X,T),
        consistent_check(T).

row_check(X,[]).
row_check(X,[Y|T]):-
        X \== Y,
        row_check(X,T).

err_counter(N, N, []).
err_counter(N, M, [err|T]):-
        N < M,
        N1 is N+1,
        err_counter(N1, M, T).

send([err]).

print_out(C,N):- nl, write('Your message is'), nl, write(C), nl,
        nl, write('The number of unmatched word is '), write(N), nl.

%% The Common Word Dictionary
%%
%% Note:
%% 1. We haven't make sure if it REALLY includes all the common words
%%    yet, ...
%% 2. Need to process word's suffixs/prefixes properly. Currently, we
%%    only have some rules like word7(A,B,C,D,i,n,g):- word4(A,B,C,D).
```

```
word1(a, []).
word1(i, []).

word2(a,m, []).
word2(a,n, []).
word2(a,t, []).
word2(b,e, []).
word2(b,y, []).
word2(d,o, []).
word2(g,o, []).
word2(h,e, []).
word2(i,f, []).
word2(i,n, []).
word2(i,s, []).
word2(i,t, []).
word2(m,e, []).
word2(m,y, []).
word2(n,o, []).
word2(o,f, []).
word2(o,n, []).
word2(o,r, []).
word2(s,o, []).
word2(t,o, []).
word2(u,p, []).
word2(w,e, []).
word2(u,s, []).

word3(a,c,t, []).
word3(a,l,l, []).
word3(a,n,y, []).
word3(a,n,d, []).
word3(a,r,e, []).
word3(c,a,n, []).
word3(d,a,y, []).
word3(d,i,d, []).
word3(f,o,r, []).
word3(h,a,d, []).
word3(h,a,s, []).
word3(h,e,r, []).
word3(h,i,m, []).
word3(h,i,s, []).
word3(h,o,w, []).
word3(i,t,s, []).
word3(m,a,n, []).
word3(m,a,y, []).
word3(m,e,n, []).
word3(n,e,w, []).
word3(n,o,t, []).
word3(n,o,w, []).
word3(o,n,e, []).
word3(o,l,d, []).
word3(o,u,r, []).
word3(s,a,w, []).
word3(s,a,y, []).
word3(s,e,e, []).
word3(s,h,e, []).
word3(s,i,x, []).
word3(t,e,n, []).
word3(t,h,e, []).
word3(t,r,y, []).
word3(t,w,o, []).
word3(w,a,s, []).
word3(w,h,o, []).
word3(w,a,s, []).
word3(y,o,u, []).
word3(X,Y,Z, [err]).
```

```
word4(a,b,l,e, []).
word4(b,e,e,n, []).
word4(b,i,l,l, []).
word4(b,o,o,k, []).
word4(b,o,t,h, []).
word4(d,o,e,s, []).
word4(d,o,n,e, []).
word4(c,a,l,l, []).
word4(c,i,t,y, []).
word4(d,a,y,s, []).
word4(d,o,w,n, []).
word4(f,a,c,t, []).
word4(f,i,v,e, []).
word4(f,o,u,r, []).
word4(f,r,o,m, []).
word4(g,a,v,e, []).
word4(g,i,v,e, []).
word4(g,o,n,e, []).
word4(g,o,e,s, []).
word4(h,a,l,f, []).
word4(h,a,v,e, []).
word4(h,i,g,h, []).
word4(i,n,t,o, []).
word4(l,a,s,t, []).
word4(l,o,v,e, []).
word4(m,a,n,y, []).
word4(m,e,e,t, []).
word4(m,o,r,e, []).
word4(n,e,x,t, []).
word4(n,i,n,e, []).
word4(o,v,e,r, []).
word4(s,a,i,d, []).
word4(s,a,y,s, []).
word4(s,e,e,s, []).
word4(s,e,e,m, []).
word4(s,e,n,d, []).
word4(s,e,n,t, []).
word4(s,o,m,e, []).
word4(s,u,c,h, []).
word4(t,a,k,e, []).
word4(t,h,a,n, []).
word4(t,h,a,t, []).
word4(t,h,e,m, []).
word4(t,h,e,y, []).
word4(t,h,i,s, []).
word4(t,i,m,e, []).
word4(t,o,o,k, []).
word4(u,p,o,n, []).
word4(v,e,r,y, []).
word4(w,h,a,t, []).
word4(w,h,o,m, []).
word4(w,i,t,h, []).
word4(w,i,l,l, []).
word4(w,h,e,n, []).
word4(y,o,u,r, []).
word4(w,a,i,t, []).
word4(w,e,n,t, []).
word4(w,e,r,e, []).
word4(y,e,a,r, []).
word4(X,Y,Z,V, [err]).

word5(a,b,o,u,t, []).
word5(a,b,o,v,e, []).
word5(a,d,o,p,t, []).
word5(a,f,t,e,r, []).
word5(a,l,o,n,g, []).
word5(a,m,o,n,g, []).
```

```
word5(a,w,a,r,e, []).
word5(b,e,i,n,g, []).
word5(b,e,l,o,w, []).
word5(b,o,o,k,s, []).
word5(c,a,l,l,s, []).
word5(c,o,u,l,d, []).
word5(e,i,g,h,t, []).
word5(f,a,c,t,s, []).
word5(f,i,r,s,t, []).
word5(g,i,v,e,n, []).
word5(g,i,v,e,s, []).
word5(l,a,r,g,e, []).
word5(l,o,v,e,d, []).
word5(l,o,v,e,s, []).
word5(m,i,g,h,t, []).
word5(m,o,n,e,y, []).
word5(o,t,h,e,r, []).
word5(p,l,a,c,e, []).
word5(r,e,a,c,h, []).
word5(s,i,n,c,e, []).
word5(s,e,e,m,s, []).
word5(s,e,v,e,n, []).
word5(s,h,a,l,l, []).
word5(s,l,e,e,p, []).
word5(s,l,e,p,t, []).
word5(t,a,k,e,s, []).
word5(t,h,e,i,r, []).
word5(t,h,i,n,k, []).
word5(t,h,r,e,e, []).
word5(t,i,m,e,s, []).
word5(t,r,a,d,e, []).
word5(t,r,i,e,s, []).
word5(t,r,i,e,d, []).
word5(u,n,d,e,r, []).
word5(w,a,t,e,r, []).
word5(w,h,i,c,h, []).
word5(w,a,i,t,s, []).
word5(w,h,i,c,h, []).
word5(w,h,o,s,e, []).
word5(w,o,u,l,d, []).
word5(w,o,m,a,n, []).
word5(w,o,m,e,n, []).
word5(y,e,a,r,s, []).
word5(X,T,Z,V,U, [err]).

word6(X1,X2,X3,i,n,g,E):- word3(X1,X2,X3,E).
word6(X1,X2,X3,X4,X5,X6,[err]).

word7(X1,X2,X3,X4,i,n,g,E):- word4(X1,X2,X3,X4,E).
word7(X1,X2,X3,X4,X5,X6,X7,[err]).

%% sub_gen(FrequencyList, AlphabetList).

sub_gen([],X).
sub_gen([A|S],X):- bind_and_delete(A,X,W), sub_gen(S,W).

bind_and_delete(X,[X|Y],Y).
bind_and_delete(A,[X|Z],[X|W]):- bind_and_delete(A,Z,W).

%% initializing an example

%% [F,P,O,U,N,M,J,B,E,S,Z,T,C,X,Q,D,G,H,Y],
%%  9 5 5 4 4 4 3 3 3 2 2 1 1 1 1 1 1 1 1

init(
 [T,F,O,E,' ',N,P,S,F,' ',N,P,O,F,Z,' ',
  U,P,' ',N,F,' ',O,F,Y,U,' ',N,F,F,U,J,O,H,
```

```
  ' ',X,J,M,M,' ',C,F,' ',P,O,' ',G,S,J,E,B,Z,
  ' ',B,U,' ',P,M,E,' ',Q,M,B,D,F],

 [N,F,' ',C,F,' ',P,O,' ',T,F,O,E,' ',
  O,F,Y,U,' ',N,P,S,F,' ',N,P,O,F,Z,' ',
  U,P,' ',B,U,' ',P,M,E,' ',X,J,M,M,' ',
  Q,M,B,D,F,' ',G,S,J,E,B,Z,' ',N,F,F,U,J,O,H],

 [F,P,O,U,N,M,J,B,E,S,Z,T,C,X,Q,D,G,H,Y],

 [F,P,O],

 [e,t,n,o,a,i,r],  %% this list could be longer

 ME ).
```

```
/* Hofstader's mu math test (mutest) proving muiiu            */

/*********************/
/* mutest            */
/* query: ?- mu_top1.*/
/*********************/

sics :- cputime(T1),
        main1,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

sics1:- cputime(T1),
        main2,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

sics2:- cputime(T1),
        main3,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

main1 :- mu_top,fail.
main1.

main2 :- mu_top1,fail.
main2.

main3 :- mu_top2,fail.
main3.

main4 :- mu_top3,fail.
main4.

mu_top :- theorem(5,[m,u,i,i,u]).
mu_top1 :- theorem(6,[m,i,i,i,i,i]).
mu_top2 :- theorem(4,[m,u,u,i]).
mu_top3 :- theorem(18,[m,i,i,i,i,i,m,u,u,u,u,u,m,i,i,i,i,i]).

rules(S,R) :- rule3(S,R).
rules(S,R) :- rule4(S,R).
```

```
rules(S,R) :- rule1(S,R).
rules(S,R) :- rule2(S,R).

rule1(S,R) :-
        app(X,[i],S),
        app(X,[i,u],R).

rule2([m|T],[m|R]) :- app(T,T,R).

rule3(R,T) :-
        app([i,i,i],S,R),
        app([u],S,T).
rule3([H|T],[H|R]) :- rule3(T,R).

rule4([H|L],T) :- app([u,u],T,[H|L]).
rule4([H|T],[H|R]) :- rule4(T,R).

theorem(_,[m,i]).
theorem(Depth,[H|L]) :-
        Depth > 0,
        D is Depth-1,
        theorem(D,S),
        rules(S,[H|L]).

app([],X,X).
app([A|B],X,[A|B1]) :-
        app(B,X,B1).




/*******************************************************

   NAVAL FLYING PROGRAM IN PANDORA

   By : Reem Bahgat

*******************************************************/
/*******************************************************

Changed to run in Prolog/Aurora by Vitor Santos Costa

*******************************************************/

/*********************/
/* flypan            */
/* queries: ?- pan2. */
/*          ?- pan4. */
/*          ?- pan5. */
/*********************/



/* The top-level procedure for this program is 'naval' .
   It accepts : (i)   a list of requirements ( i.e. flights ) to be scheculed ,
                (ii)  a list of available resources ( i.e. aircraft ) , and
                (iii) an integer value giving the maximum number of hours
                      of work for the members of crew
   The output consists of a list of tuples , each one specifying the aircraft
   that has been allocated to each flight in the input requirements list .
*/

sics1:- cputime(T1),
        main1,
        cputime(T2),
        T is T2 - T1,
```

```
        write('Time: '),
        write(T),nl.

main1:- pan2,
        fail.
main1.

sics2:- cputime(T1),
        main2,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

main2:- pan3,
        fail.
main2.

sics3:- cputime(T1),
        main3,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

main3:- pan4,
        fail.
main3.

sics4:- cputime(T1),
        main4,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

main4:- pan5,
        fail.
main4.

sics5:- cputime(T1),
        main5,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

main5:- pan6,
        fail.
main5.

sics6:- cputime(T1),
        main6,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

main6:- pan7,
        fail.
main6.

sics7:- cputime(T1),
        main7,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
```

```
        write(T),nl.

main7:- pan8,
        fail.
main7.

sics8:- cputime(T1),
        main8,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

main8:- pan9,
        fail.
main8.

sics9:- cputime(T1),
        main9,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

main9:- pan10,
        fail.
main9.

pan1 :-
        A=[flight(1,2,10,bo707,3),
            flight(2,4,9,bo707,3),
            flight(3,11,15,bo707,5),
           flight(5,10,23,bo707,6),
           flight(6,2,10,bo747,4),
           flight(7,5,6,dc10,3),
           flight(8,6,21,dc10,4),
            flight(4,16,19,helicopter,2),
           flight(9,7,23,concord,10),
           flight(10,3,19,concord,7)],
        B=[aircraft(1,bo707,12,5,2,12,[c1,c2,c3,c4,c5]),
           aircraft(2,bo707,10,3,1,15,[c1,c6,c7]),
           aircraft(3,bo707,7,5,1,8,[c8,c9,c10,c11,c12]),
           aircraft(4,bo707,5,2,10,4,[c3,c4,c6,c7]),
           aircraft(5,bo747,12,1,18,10,[c1,c2,c3,c4,c5,c6,c7,c8,c9,c10]),
           aircraft(6,helicopter,3,2,1,10,[c4,c9]),
           aircraft(7,dc10,8,2,6,3,[c5,c7,c9]),
           aircraft(8,dc10,6,2,10,2,[c3,c4]),
           aircraft(9,concord,20,3,24,7,[c1,c3,c5,c6,c7,c8,c10]),
           aircraft(10,concord,24,5,26,8,[c1,c2,c3,c4,c5,c6,c7,c8]),
           aircraft(11,bo747,10,3,20,4,[c2,c4,c6,c8])],
        naval(A,B,15,C),
        write(C),
        nl.

pan2 :-
        A=[flight(1,2,10,bo707,3),
            flight(2,4,9,bo707,3),
            flight(3,11,15,bo707,5),
           flight(4,10,17,bo707,6),
           flight(5,2,10,bo747,4),
            flight(6,16,19,helicopter,2),
           flight(7,5,6,dc10,3),
           flight(8,6,21,dc10,4),
           flight(9,7,23,concord,8),
           flight(10,3,5,concord,7)],
        B=[aircraft(1,bo707,12,5,1,20,[c1,c2,c3,c4,c5]),
           aircraft(2,bo707,10,3,2,10,[c1,c2,c3]),
```

```
           aircraft(3,bo707,8,6,1,16,[c6,c7,c8,c9,c10,c11]),
           aircraft(4,bo707,9,5,3,9,[c3,c4,c6,c7,c8]),
           aircraft(5,bo747,12,10,2,18,
                        [c12,c13,c14,c15,c16,c17,c18,c19,c20,c21]),
           aircraft(6,helicopter,3,2,1,10,[c1,c2]),
           aircraft(7,dc10,16,4,2,16,[c41,c51,c22,c23]),
           aircraft(8,dc10,16,5,1,16,[c24,c25,c26,c27,c28]),
           aircraft(9,concord,20,7,1,24,
                        [c411,c511,c221,c231,c291,c301,c311]),
           aircraft(10,concord,24,8,2,26,
                        [c29,c30,c31,c32,c33,c34,c35,c36]),
           aircraft(11,bo747,10,4,2,20,[c29,c30,c31,c32])],
        naval(A,B,20,C)/*,
        write(C),
        nl*/.

pan3 :-
        A=[flight(1,2,10,bo707,3),
            flight(2,4,9,bo707,3),
            flight(3,11,15,bo707,5),
           flight(4,10,17,bo707,6),
           flight(5,2,10,bo747,4),
            flight(6,16,19,helicopter,2),
           flight(7,5,6,dc10,3),
           flight(8,6,21,dc10,4),
           flight(9,7,23,concord,8),
           flight(10,3,5,concord,7)],
        B=[aircraft(1,bo707,12,5,1,20,[c1,c2,c3,c4,c5]),
           aircraft(2,bo707,10,3,2,10,[c6,c7,c8]),
           aircraft(3,bo707,8,6,1,16,[c9,c10,c11,c12,c13,c14]),
           aircraft(4,bo707,9,4,3,9,[c15,c16,c17,c18]),
           aircraft(5,bo747,12,10,2,18,
                        [c19,c20,c21,c22,c23,c24,c25,c26,c27,c28]),
           aircraft(6,helicopter,3,2,1,10,[c29,c30]),
           aircraft(7,dc10,16,4,2,16,[c31,c32,c33,c34]),
           aircraft(8,dc10,16,5,1,16,[c35,c36,c37,c38,c39]),
           aircraft(9,concord,20,7,1,24,[c40,c41,c42,c43,c44,c45,c46]),
           aircraft(10,concord,24,8,2,26,[c47,c48,c49,c50,c51,c52,c53,c54]),
           aircraft(11,bo747,10,4,2,20,[c55,c56,c57,c58])],
        naval(A,B,20,C)/*,
        write(C),
        nl*/.

pan4 :-
        A=[flight(1,2,10,bo707,3),
            flight(2,4,9,bo707,3),
            flight(3,11,15,bo707,5),
           flight(4,10,17,bo707,6),
           flight(5,2,10,bo747,4),
            flight(6,16,19,helicopter,2),
           flight(7,5,6,dc10,3),
           flight(8,6,21,dc10,4),
           flight(9,7,23,concord,8),
           flight(10,3,5,concord,7)],
        B=[aircraft(1,bo707,12,5,1,20,[c1,c2,c3,c4,c5]),
           aircraft(2,bo707,10,3,2,10,[c1,c2,c3]),
           aircraft(3,bo707,8,6,1,16,[c6,c7,c8,c9,c10,c11]),
           aircraft(4,bo707,9,5,3,9,[c3,c4,c6,c7,c8]),
           aircraft(5,bo747,12,10,2,18,
                        [c12,c13,c14,c15,c16,c17,c18,c19,c20,c21]),
           aircraft(6,helicopter,3,2,1,10,[c1,c2]),
           aircraft(7,dc10,16,4,2,16,[c41,c51,c22,c23]),
           aircraft(8,dc10,16,5,1,16,[c24,c25,c26,c27,c28]),
           aircraft(9,concord,20,7,1,24,
                        [c411,c511,c221,c231,c291,c301,c311]),
           aircraft(10,concord,24,8,2,26,
                        [c29,c30,c31,c32,c33,c34,c35,c36]),
```

```
            aircraft(11,bo747,10,4,2,20,[c29,c30,c31,c32]),
            aircraft(12,dc10,15,4,1,16,[c25,c26,c27,c28]),
            aircraft(13,dc10,20,5,1,16,[c25,c26,c27,c28,c29]),
            aircraft(14,bo747,12,4,2,20,[c68,c69,c70,c71])],
         naval(A,B,20,C)/*,
         write(C),
         nl*/.

pan5 :-
         A=[flight(1,2,10,bo707,3),
             flight(2,4,9,bo707,3),
             flight(3,11,15,bo707,5),
            flight(4,10,17,bo707,6),
            flight(5,2,10,bo747,4),
             flight(6,16,19,helicopter,2),
            flight(7,5,6,dc10,3),
            flight(8,6,21,dc10,4),
            flight(9,7,23,concord,8),
            flight(10,3,5,concord,7)],
         B=[aircraft(1,bo707,12,5,1,20,[c1,c2,c3,c4,c5]),
            aircraft(2,bo707,10,3,2,10,[c6,c7,c8]),
            aircraft(3,bo707,8,6,1,16,[c9,c10,c11,c12,c13,c14]),
            aircraft(4,bo707,9,4,3,9,[c15,c16,c17,c18]),
            aircraft(5,bo747,12,10,2,18,
                          [c19,c20,c21,c22,c23,c24,c25,c26,c27,c28]),
            aircraft(6,helicopter,3,2,1,10,[c29,c30]),
            aircraft(7,dc10,16,4,2,16,[c31,c32,c33,c34]),
            aircraft(8,dc10,16,5,1,16,[c35,c36,c37,c38,c39]),
            aircraft(9,concord,20,7,1,24,
                          [c40,c41,c42,c43,c44,c45,c46]),
            aircraft(10,concord,24,8,2,26,
                          [c47,c48,c49,c50,c51,c52,c53,c54]),
            aircraft(11,bo747,10,4,2,20,[c55,c56,c57,c58]),
            aircraft(12,dc10,15,4,1,16,[c59,c60,c61,c62]),
            aircraft(13,dc10,20,5,1,16,[c63,c64,c65,c66,c67]),
            aircraft(14,bo747,12,3,2,20,[c68,c69,c70])],
         naval(A,B,20,C)/*,
         write(C),
         nl*/.

pan6 :-
         A=[flight(1,2,10,bo707,3),flight(2,4,9,bo707,3),
                flight(3,11,15,bo707,5),flight(4,16,19,helicopter,2)],
         B=[aircraft(1,bo707,12,5,2,12,[c1,c2,c3,c4,c5]),
                aircraft(2,bo707,10,3,1,15,[c1,c6,c7]),
                aircraft(3,bo707,7,5,1,8,[c8,c9,c10,c11,c12]),
                aircraft(4,helicopter,3,2,1,10,[c4,c9])],
         naval(A,B,15,C)/*,
         write(C),
         nl*/.

pan8 :-
         A=[flight(1,2,10,bo707,3),
             flight(2,4,9,bo707,3),
             flight(3,11,15,bo707,5),
            flight(4,10,17,bo707,6),
            flight(5,2,10,bo747,4),
             flight(6,16,19,helicopter,2),
            flight(7,5,6,dc10,3),
            flight(8,6,21,dc10,4),
            flight(9,7,23,concord,8),
            flight(10,3,5,concord,7)],
         B=[aircraft(1,bo707,12,5,1,20,[c1,c2,c3,c4,c5]),
            aircraft(2,bo707,10,3,2,10,[c1,c2,c3]),
            aircraft(3,bo707,8,6,1,16,[c6,c7,c8,c9,c10,c11]),
            aircraft(4,bo707,9,5,3,9,[c3,c4,c6,c7,c8]),
            aircraft(5,bo747,12,10,2,18,
```

```
                    [c12,c13,c14,c15,c16,c17,c18,c19,c20,c21]),
          aircraft(6,helicopter,3,2,1,10,[c1,c2]),
          aircraft(7,dc10,16,4,2,16,[c41,c51,c22,c23]),
          aircraft(8,dc10,16,5,1,16,[c24,c25,c26,c27,c28]),
          aircraft(9,concord,20,7,1,24,
                    [c411,c511,c221,c231,c291,c301,c311]),
          aircraft(10,concord,24,8,2,26,
                    [c29,c30,c31,c32,c33,c34,c35,c36]),
          aircraft(11,bo747,10,4,2,20,[c29,c30,c31,c32])],
/*        aircraft(12,dc10,15,4,1,16,[c25,c26,c27,c28]),
          aircraft(13,dc10,20,5,1,16,[c25,c26,c27,c28,c29]),
          aircraft(14,bo747,12,4,2,20,[c68,c69,c70,c71])],*/
        naval(A,B,20,C)/*,
        write(C),
        nl*/.

pan9 :-
        A=[flight(1,2,10,bo707,3),
           flight(2,4,9,bo707,3),
           flight(3,11,15,bo707,5),
           flight(4,10,17,bo707,6),
           flight(5,2,10,bo747,4),
           flight(6,16,19,helicopter,2),
           flight(7,5,6,dc10,3),
           flight(8,6,21,dc10,4),
           flight(9,7,23,concord,8),
           flight(10,3,5,concord,7)],
        B=[aircraft(1,bo707,12,5,1,20,[c1,c2,c3,c4,c5]),
          aircraft(2,bo707,10,3,2,10,[c1,c2,c3]),
          aircraft(3,bo707,8,6,1,16,[c6,c7,c8,c9,c10,c11]),
          aircraft(4,bo707,9,5,3,9,[c3,c4,c6,c7,c8]),
          aircraft(5,bo747,12,10,2,18,
                    [c12,c13,c14,c15,c16,c17,c18,c19,c20,c21]),
          aircraft(6,helicopter,3,2,1,10,[c1,c2]),
          aircraft(7,dc10,16,4,2,16,[c41,c51,c22,c23]),
          aircraft(8,dc10,16,5,1,16,[c24,c25,c26,c27,c28]),
          aircraft(9,concord,20,7,1,24,
                    [c411,c511,c221,c231,c291,c301,c311]),
          aircraft(10,concord,24,8,2,26,
                    [c29,c30,c31,c32,c33,c34,c35,c36]),
          aircraft(11,bo747,10,4,2,20,[c29,c30,c31,c32]),
          aircraft(12,dc10,15,4,1,16,[c25,c26,c27,c28])],
/*        aircraft(13,dc10,20,5,1,16,[c25,c26,c27,c28,c29]),
          aircraft(14,bo747,12,4,2,20,[c68,c69,c70,c71])],*/
        naval(A,B,20,C)/*,
        write(C),
        nl*/.

pan10 :-
        A=[flight(1,2,10,bo707,3),
           flight(2,4,9,bo707,3),
           flight(3,11,15,bo707,5),
           flight(4,10,17,bo707,6),
           flight(5,2,10,bo747,4),
           flight(6,16,19,helicopter,2),
           flight(7,5,6,dc10,3),
           flight(8,6,21,dc10,4),
           flight(9,7,23,concord,8),
           flight(10,3,5,concord,7)],
        B=[aircraft(1,bo707,12,5,1,20,[c1,c2,c3,c4,c5]),
          aircraft(2,bo707,10,3,2,10,[c1,c2,c3]),
          aircraft(3,bo707,8,6,1,16,[c6,c7,c8,c9,c10,c11]),
          aircraft(4,bo707,9,5,3,9,[c3,c4,c6,c7,c8]),
          aircraft(5,bo747,12,10,2,18,
                    [c12,c13,c14,c15,c16,c17,c18,c19,c20,c21]),
          aircraft(6,helicopter,3,2,1,10,[c1,c2]),
          aircraft(7,dc10,16,4,2,16,[c41,c51,c22,c23]),
```

```
           aircraft(8,dc10,16,5,1,16,[c24,c25,c26,c27,c28]),
           aircraft(9,concord,20,7,1,24,
                        [c411,c511,c221,c231,c291,c301,c311]),
           aircraft(10,concord,24,8,2,26,
                        [c29,c30,c31,c32,c33,c34,c35,c36]),
           aircraft(11,bo747,10,4,2,20,[c29,c30,c31,c32]),
           aircraft(12,dc10,15,4,1,16,[c25,c26,c27,c28]),
           aircraft(13,dc10,20,5,1,16,[c25,c26,c27,c28,c29])],
/*         aircraft(14,bo747,12,4,2,20,[c68,c69,c70,c71])],*/
        naval(A,B,20,C)/*,
        write(C),
        nl*/.

main :-
        Flights = [flight(1, 2, 10, bo707, 3),
           flight(2, 4, 9, bo707, 3),
           flight(3, 11, 15, bo707, 5),
           flight(4, 16, 19, helicopter, 2)],
           Aircraft = [aircraft(1,bo707,12,5,2,12,[c1, c2, c3, c4, c5]),
            aircraft(2, bo707, 10, 3, 1, 15, [c1, c6, c7]),
            aircraft(3, bo707, 7, 5, 1, 8, [c8, c9, c10, c11, c12]),
            aircraft(4, helicopter, 3, 2, 1, 10, [c4, c9])],
            naval(Flights, Aircraft, 15, Assignments),
            write(Assignments), nl.


pan7 :-
        Flights = [flight(1, 2, 10, bo707, 3),
           flight(2, 4, 9, bo707, 3),
           flight(3, 11, 15, bo707, 5),
           flight(4, 16, 19, helicopter, 2)],
           Aircraft = [aircraft(1, bo707,12,5,2,12,[c1, c2, c3, c4, c5]),
            aircraft(2, bo707, 10, 3, 1, 15, [c1, c6, c7]),
            aircraft(3, bo707, 7, 5, 1, 8, [c8, c9, c10, c11, c12]),
            aircraft(4, helicopter, 3, 2, 1, 10, [c4, c9]),
            aircraft(5, bo707, 12, 5, 2, 12, [c1, c2, c3, c4, c5]),
            aircraft(6, bo707, 10, 3, 1, 15, [c1, c6, c7]),
            aircraft(7, bo707, 7, 5, 1, 8, [c8, c9, c10, c11, c12]),
            aircraft(8, helicopter, 3, 2, 1, 10, [c4, c9])],
            naval(Flights, Aircraft, 15, Assignments).

:- mode naval(+,+,+,-).
naval( Flights, Aircraft, Max_crew_hour, Assignments )
     :- assign_flights( Flights, Aircraft, Max_crew_hour, Ons ) ,
        extract_assignments( Ons, Assignments ) .




/* The next two procedures provide a simple interface with the user .
   The intermediate output of the program consists of a list of the
   structures of the given flights . Each element of the list would
   therefore consist of a flight and a list of tuples specifying
   the aircraft that are 'off' for that flight , and the aircraft
   that is 'on' for the flight .
   The final output produced by 'extract_assignments' and
   'find_aircraft' consists of a list of tuples , where each tuple
   combines a flight with the aircraft that was allocated to it after
   the end of the execution .
*/

:- mode extract_assignments(+,-).
extract_assignments( [poss_aircraft(F,Aircraft)|Ons], [AircraftForF|Rest])
     :- find_aircraft( F, Aircraft, AircraftForF ) ,
        extract_assignments( Ons, Rest ) .
extract_assignments( [], [] ) .
```

```
:- mode find_aircraft(+,+,-).
find_aircraft( F, [value(A1,on)|Ons1], aircraft_for_flight( F_no, A_no ) )
    :- functor(F,flight,_), arg(1,F,F_no), % F =.. [flight, F_no|Flight],
       functor(A1,aircraft,_), arg(1,A1,A_no).
       % A1 =.. [aircraft, A_no|Aircraft].
find_aircraft( F, [value(A1,off)|Ons1], Assigned_aircraft )
    :- find_aircraft( F, Ons1, Assigned_aircraft ) .




/* 'assign_flights' is the main procedure in this program .
    It generates the structure of the flights , whereby each flight
   has associated with it a series of aircraft . Each aircraft has
   a unique variable associated with it .
   The given constraints are called in parallel from this procedure .
   3 types of constraints:
   1. unary constraints, applied on each flight structure independent of other
      flights.
   2. binary constraints applied on each 2 flight structures.
   3. plural constraints related to each aircraft/crew and thus applied on
      several flights.
   The procedure terminates when each flight is allocated an aircraft.
*/


:- mode assign_flights(+,+,+,-).
assign_flights( Flights, Aircraft, Max_crew_hour, By_flights )
    :- single_flight_constraints(Flights, Aircraft, By_flights),
       arrange_by_crew(Aircraft, By_crew),
       two_flight_constraints( By_flights, By_crew ),
       arrange_by_aircraft(Aircraft, By_flights, By_aircraft),
       many_flight_constraints( By_aircraft, By_crew, Max_crew_hour).




/* The required attributes of a flight are extracted by 'flight_attrib'
   and passed to 'unary_constr',
   all those aircraft that do not satisfy the single flight constraints are
   removed from the flight's domain . This is done by assigning the value 'off'
   to the variables associated with such aircraft .
   Single flight constraints are:

   1. The Type Constraint: states that a flight can only be allocated
      an aircraft if that aircraft has the correct type.
   2. The Endurance Constraint: states that an aircraft cannot fly longer than
      its endurance permits continuously.
   3. The Crew Constraint: states that there must be sufficient crew on an
      aircraft to fly a requirement .
*/


:- mode single_flight_constraints(+, +, -).
single_flight_constraints([], _, []).
single_flight_constraints([F1|Fs], Aircraft,
                          [poss_aircraft(NewF1,Ons1)|By_flights])
    :-  generate_aircraft(Poss_aircraft, A, begin, end, Ons1),
        single_flight_constraint(F1, Aircraft, Poss_aircraft, NewF1),
       single_flight_constraints(Fs, Aircraft, By_flights).


:- mode single_flight_constraint( + , +, -, -).
```

```
single_flight_constraint( F, Aircraft, Poss_aircraft, NewF )
    :- flight_attrib(F, NewF, Attributes),
       unary_constrs( Aircraft, Attributes, Poss_aircraft).

:- mode flight_attrib(+, -, -).
flight_attrib( flight( No, Launch, Recovery,Type,Crew),
               flight( No, Launch, Recovery, Length),
               attrib(Type, Crew, Length) )
    :- Length is Recovery - Launch.


:- mode unary_constrs(+, +, -).
unary_constrs( [], _, [] ) .
unary_constrs([A|Aircraft], Attributes, [NewA|Poss_aircraft] )
   :- A = aircraft(No,Type,Endurance,AvailCrew,Turnaround,MaxHrs,Crew_list),
%correct_attrib( A, Attributes, NewA), !,
       Attributes = attrib(Type, Crew, Flight_length),
       AvailCrew >= Crew, Endurance >= Flight_length, !,
       NewA = aircraft(No,Turnaround,MaxHrs,Crew_list),
        unary_constrs( Aircraft, Attributes, Poss_aircraft).
unary_constrs( [A|Aircraft], Attributes, Poss_aircraft)
   :- unary_constrs( Aircraft, Attributes, Poss_aircraft).

:- mode correct_attrib(+, +, -).
correct_attrib(aircraft(No,Type,Endurance,AvailCrew,Turnaround,
                                             MaxHrs,Crew_list),
               attrib(Type, Crew, Flight_length),
               aircraft(No,Turnaround,MaxHrs,Crew_list))
    :- AvailCrew >= Crew, Endurance >= Flight_length.


/* 'generate_aircraft' forms a chain for a particular flight consisting
   of the aircraft in the input list of available resources , where each
   aircraft has associated with it a unique variable that may be still
   uninstantiated , or may have a value 'on' or 'off' . If an aircraft
   is 'on' for a flight , it means that the aircraft is allocated to the
   flight . An aircraft that is 'off' for a flight has been removed from
   flight's possible set .

   A Prolog procedure for the generation of the cell items representing
   the aircraft is given here . It would be used in the non-deterministic
   phase of the execution to allocate an aircraft to a flight .

   A deterministic Parlog procedure for the cell items is also given .
   The clauses in this procedure would be used in the deterministic phase,
   when an aircraft may be allocated deterministically to a flight , or an
   aircraft may be removed deterministically from the flight's domain .
*/

item( Aircraft, Aircraft, begin, end, on ).
item( _, _, Ch, Ch, off ).


:- mode generate_aircraft(+,+,-,+,-).
generate_aircraft( [], _, Right, Right, [] ).
generate_aircraft( [A1|Aircraft], Flight, Left, Right, [value(A1,On1)|Ons] )
    :- item( A1, Flight, Left, Temp, On1 ) ,
       generate_aircraft( Aircraft, Flight, Temp, Right, Ons ).


/*
:- mode item(+,+,+,+,+).
item( Aircraft, A, Left, Right, On )
    :- A \== Aircraft :
       Left = Right ,
       On = off .
item( Aircraft, A, begin, end, On )
```

```
    :- A = Aircraft ,
       On = on .
item( Aircraft, A, Left, Right, on )
    :- A = Aircraft ,
       Left = begin ,
       Right = end .
item( Aircraft, A, Left, Right, off )
    :- Left = Right .

*/



/* arrange_by_crew takes the list of aircraft and produces a list of crew,
  each element in the list is a list of aircraft that are sharing the crew
  member represented by this element.
*/


:- mode arrange_by_crew(+, +).
arrange_by_crew(Aircraft, By_crew)
    :- extract_crew_list(Aircraft, Tuples, Crew_list, []),
       group_by_crew(Tuples, Crew_list, By_crew).


/* extract_crew_list returns a list of tuples: (Aircraft_no, Crew), and
  a list of the different available crew.
*/

:- mode extract_crew_list(+, -, -, +).
extract_crew_list([], [], Crew_list, Crew_list).
extract_crew_list([aircraft(A_no,_,_,_,_,_,Crew1)|Aircraft],
                  [aircraft(A_no,Crew1)|Tuples], Crew_list, SoFar)
    :- add_to_list(Crew1, SoFar, New_list),
       extract_crew_list(Aircraft, Tuples, Crew_list, New_list).

:- mode add_to_list(+, +, -).
add_to_list([], Crew_list, Crew_list).
/*add_to_list([C1|Crew], SoFar, Crew_list)
    :- member(C1, SoFar), !,
       add_to_list(Crew, SoFar, Crew_list).
add_to_list([C1|Crew], SoFar, [C1|Crew_list])
    :- add_to_list(Crew, SoFar, Crew_list).
*/
add_to_list([C1|Crew], SoFar, Crew_list) :-
        add_to_list_2([C1|Crew], SoFar, Crew_list,SoFar).


:- mode add_to_list_2(+, +, -, +).
add_to_list_2([C1|Crew], SoFar, Crew_list, [C1|_]) :- !,
        add_to_list(Crew, SoFar, Crew_list).
add_to_list_2([C1|Crew], SoFar, Crew_list, [_|SoFarT]) :- !,
        add_to_list_2([C1|Crew], SoFar, Crew_list, SoFarT).
add_to_list_2([C1|Crew], SoFar, [C1|Crew_list], _) :-
        add_to_list(Crew, SoFar, Crew_list).


:- mode member(+, +).
member(X, [X|_]) :- !.
member(X, [Y|Ys])
  :- member(X, Ys).


/* for each crew get the list of aircraft which he is member of. The output
   is a list of these lists.
*/
```

```
:- mode group_by_crew(+, +, -).
group_by_crew(_, [], []).
group_by_crew(Tuples, [C1|Crew], [By_C1|By_crew])
    :- for_one_crew(Tuples, C1, By_C1),
       group_by_crew(Tuples, Crew, By_crew).

:- mode for_one_crew(+, +, -).
for_one_crew([], _, []).
/*
for_one_crew([aircraft(A_no,Crew_list)|Tuples], Crew, [A_no|By_crew])
    :- member(Crew, Crew_list), !,
       for_one_crew(Tuples, Crew, By_crew).
for_one_crew([A1|Aircraft], Crew, By_crew)
    :- for_one_crew(Aircraft, Crew, By_crew).
*/
for_one_crew([aircraft(A_no,[Crew|_])|Tuples], Crew, [A_no|By_crew]) :- !,
        for_one_crew(Tuples, Crew, By_crew).
for_one_crew([aircraft(A_no,[_|CrewList])|Tuples], Crew, By_crew) :- !,
        for_one_crew([aircraft(A_no,CrewList)|Tuples], Crew, By_crew).
for_one_crew([A1|Aircraft], Crew, By_crew)
    :- for_one_crew(Aircraft, Crew, By_crew).



/* If two flights overlap in time , they cannot be assigned the same
   aircraft, or different aircraft with a shared crew.

   If two flights do not overlap , then the Turnaround Constraint would be
   applied to their domains, no need to apply it on overlapping ones .
*/

:- mode two_flight_constraints(+, +).
two_flight_constraints([], _).
two_flight_constraints([F1|Flights], By_crew)
    :- check(F1, Flights, By_crew),
       two_flight_constraints(Flights, By_crew).

:- mode check(+,+, +).
check( _, [], _ ).
check(F1, [F2|Flights], By_crew)
    :- binary_constrs(F1, F2, By_crew),
       check(F1, Flights, By_crew).

:- mode binary_constrs(+, +, +).
binary_constrs(poss_aircraft(F1, A1), poss_aircraft(F2, A2), By_crew)
%    :- overlap(F1, F2), !,
        :- F1 = flight(Flight_no1, _, Recovery_time1, _),
        F2 = flight(Flight_no2,Launch_time2, _, _),
        Recovery_time1 > Launch_time2, !,
       noteq_aircraft(A1, A2),
       noteq_crew(By_crew, A1, A2).
binary_constrs(poss_aircraft(F1,A1), poss_aircraft(F2,A2), _)
    :- turnaround_constraint(F1, A1, F2, A2).



/* 'overlap' is true if the two flights that are input to it overlap
   in time .
*/


:- mode overlap( +, + ).
overlap( flight(Flight_no1, _, Recovery_time1, _),
         flight(Flight_no2,Launch_time2, _, _) )
    :- Recovery_time1 > Launch_time2.
```

```
/* 'noteq_aircraft' accepts two lists representing the domains of
   two flights , and succeeds if the same aircraft in the two domains
   satisfy the noteq constraint .

   The inputs to the 'noteq' procedure are the two variables associated
   with two aircraft . If one of them is 'on' , the other is forced to
   be 'off' . A variable that is found to be 'off' does not affect the
   value of the second variable .
*/


:- mode noteq_aircraft(+,+).
noteq_aircraft( [], _ ) :- !.
noteq_aircraft( _, [] ) :- !.
noteq_aircraft( [value(A,On1)|Ons1], [value(A,On2)|Ons2] )
    :- !, noteq( On1, On2 ),
       noteq_aircraft( Ons1, Ons2 ).
noteq_aircraft( [value(A1,_)|Ons1], [value(A2,On2)|Ons2])
%:- aircraft_less(A1, A2), !,
        :- functor(A1,aircraft,_), arg(1,A1,A_no1),
        functor(A2,aircraft,_), arg(1,A2,A_no2),
        A_no1 < A_no2, !,
       noteq_aircraft(Ons1, [value(A2,On2)|Ons2]).
noteq_aircraft( [A1|Ons1], [A2|Ons2])
    :- noteq_aircraft( [A1|Ons1], Ons2).


/****************

For Prolog

/****************/

:- mode noteq(+,+).
noteq( on, On2 )
    :- ! , On2 = off .
noteq( On1, on )
    :- ! , On1 = off .
noteq( off, _ ) :- ! .
noteq( _, off ) :- ! .


/***************
:- mode noteq(+,+).
noteq( on, On2 )
    :- / , On2 = off .
noteq( On1, on )
    :- / , On1 = off .
noteq( off, _ ) :- / .
noteq( _, off ) :- / .

****************/

:- mode aircraft_less(+, +).
aircraft_less(A1, A2)
    :- functor(A1,aircraft,_), arg(1,A1,A_no1),
       % A1 =.. [aircraft, A_no1|Args1],
       functor(A2,aircraft,_), arg(1,A2,A_no2),
       % A2 =.. [aircraft, A_no2|Args2],
       A_no1 < A_no2.
```

```
/* noteq_crew accepts 3 inputs:
    1. a list of lists, each consists of the aircraft that are sharing a
       particular crew, e.g. [ [1,2,4], [2,3,4]]
    2. 2 flights' domains that are overlapping
   The given overlapping flights are prevented from being assigned different
   aircraft with a common crew.
*/

:- mode noteq_crew(+, +, +).
noteq_crew([], _, _).
noteq_crew([Crew1|Crew], F1, F2)
    :- check_one_crew(Crew1, F1, F2),
       noteq_crew(Crew, F1, F2).



/* For aircraft with a common crew, for each aircraft A1 in this list, A1 in
   flight F1 is connected to all other aircraft of the list that they exist
   in flight F2 with noteq constraint so that if A1 is assigned to F1, no
   one from the other aircraft can be assigned to F2. Similarly, A1 in F2 is
   connected to the other aircraft from F1. This process is repeated among
   the other aircraft in the list.

   Because, aircraft in a flight's domain are arranged increasingly
   according to their A_no, to find aircraft from the domain, we can always
   start looking from where the previous aircraft was found instead of
   looking each time from the begining.
*/

:- mode check_one_crew(+, +, +).
check_one_crew([], _, _).
check_one_crew([A1|Aircraft], F1, F2)
    :- get_aircraft(A1, F1, On1, RestF1),
       each_aircraft(Aircraft, F2, On1),
       get_aircraft(A1, F2, On2, RestF2),
       each_aircraft(Aircraft, F1, On2),
       check_one_crew(Aircraft, RestF1, RestF2).



/* get the On variable of the eaircraft A_no from the flight's domain.
   Also output the rest of aircraft in the domain.
*/

:- mode get_aircraft(+, +, -, -).
get_aircraft(A_no, [value(aircraft(A_no,_,_,_),On1)|Aircraft],
             On1, Aircraft) :- !.
get_aircraft(A_no, [value(aircraft(A_no2,_,_,_),_)|Aircraft],
             On1, Rest)
    :- A_no > A_no2, !,
       get_aircraft(A_no, Aircraft, On1, Rest).
get_aircraft(_, Aircraft, off, Aircraft).



/* connect the aircraft 'On2' with all aircraft in flight F1 and sharing a
   crew with 'On2'.  */


/*************

For Prolog:

*****************/

:- mode each_aircraft(+, +, +).
each_aircraft(_, _, off) :- ! .
```

```
each_aircraft([], _, _) :- ! .
each_aircraft(_, [], _) :- ! .
each_aircraft([A|Aircraft], F1, On2)
    :- ! , get_aircraft(A, F1, On1, RestF1),
       noteq(On1, On2),
       each_aircraft(Aircraft, RestF1, On2).


/******************
:- mode each_aircraft(+, +, +).
each_aircraft(_, _, off) :- / .
each_aircraft([], _, _) :- / .
each_aircraft(_, [], _) :- / .
each_aircraft([A|Aircraft], F1, On2)
    :- / , get_aircraft(A, F1, On1, RestF1),
       noteq(On1, On2),
       each_aircraft(Aircraft, RestF1, On2).

*******************/


/* The Turnaround Constraint states that there must be sufficient time
   between flights of the same aircraft to turn around the aircraft .

   The procedure receives two flights with the representations associated
   with their domains . The turnaround time of each aircraft in the domain
   is used to calculate the next possible launch time for any flight that
   might want to use that aircraft after the aircraft was allocated to the
   flight given in the first argument . This calulated value corresponds to
   the sum of the recovery time of the first flight and the turnaround time
   of the aircraft . If the second flight's launch time is before this value
   , the 'noteq' constraint would be applied to the variables associated
   with that aircraft for the two flights .  */


:- mode turnaround_constraint( +,+,+,+ ).
turnaround_constraint( _, [], _, _ ) :- !.
turnaround_constraint( _, _, _, [] ) :- !.
turnaround_constraint( F1, [value(A1,On1)|Aircraft1],
                       F2, [value(A1,On2)|Aircraft2] )
    :- !, get_next_launch(F1, A1, Next_launch),
       check_turnaround(Next_launch, F2, On1, On2),
       turnaround_constraint( F1, Aircraft1, F2, Aircraft2 ).
turnaround_constraint( F1, [value(A1,_)|Aircraft1],
                       F2, [value(A2,On2)|Aircraft2] )
%    :- aircraft_less(A1, A2), !,
        :- functor(A1,aircraft,_), arg(1,A1,A_no1),
        functor(A2,aircraft,_), arg(1,A2,A_no2),
        A_no1 < A_no2, !,
       turnaround_constraint(F1, Aircraft1, F2, [value(A2,On2)|Aircraft2]).
turnaround_constraint( F1, [value(A1,On1)|Aircraft1],
                       F2, [value(A2,_)|Aircraft2] )
    :- turnaround_constraint(F1, [value(A1, On1)|Aircraft1], F2, Aircraft2).


:- mode get_next_launch(+, +, -).
get_next_launch( flight( _, _,Recovery_time1, _),
                 aircraft( _, Turnaround, _, _), Next_launch )
    :- Next_launch is Recovery_time1 + Turnaround .



:- mode check_turnaround(+,+,+,+).
check_turnaround( Next_launch,
                  flight( _, Launch_time, _, _), On1, On2 )
    :- Next_launch > Launch_time, !,
       noteq( On1, On2 ).
```

```
check_turnaround( _, _, _, _).

/* 'arrange_by_aircraft' takes the list of available aircraft, and
   the list of flights' domains, and produces a list representing aircraft
   that are at least in one flight's domain. Each aircraft in the ouput is
   a term of the form: aircraft(A_no, MaxHours, Flights). 'Flights' is a
   list of tuples of the form:tup(Flight_length, ON), representing a flight
   which the aircraft A_no is a member of its domain.

  For each available aircraft, get the flights which it is a member of their
  domains. If the aircraft is not a member in any flight's domain, don't
  generate an element for it in the output.

  Since aircraft are ordered in increasing order according to thei A_no,
  don't fetch the aircraft in a flight's domain from the begining of the
  domain. Instead, start from where the previous aircraft was found.
*/



:- mode arrange_by_aircraft(+, +, -).
arrange_by_aircraft([], _, []) :- !.
arrange_by_aircraft(_, [], []) :- !.
arrange_by_aircraft([A1|Aircraft], Flights, By_aircraft)
    :- get_tuples(Flights, A1, Tuples, NewFlights),
       check_tuples(Tuples, A1, By_aircraft, By_Tail),
       arrange_by_aircraft(Aircraft, NewFlights, By_Tail).



:- mode check_tuples(+, +, -, +).
check_tuples([], _, Tail, Tail) :- !.
check_tuples(Tuples, Aircraft, [aircraft(A_no,MaxHr,Tuples)|Tail], Tail)
    :- aircraft_attrib(Aircraft, A_no, MaxHr).

:- mode aircraft_attrib(+, -, -).
aircraft_attrib(aircraft(A_no,_,_,_,_,MaxHrs,_), A_no, MaxHrs).



/* get tuples of flights in which this aircraft is a member, and return the
   remaining of the flights domains.
*/



:- mode get_tuples(+, +, -, -).
get_tuples([], _, [], []).
get_tuples([poss_aircraft(F,A)|Flights], Aircraft, Tuples, NewFlights)
    :- get_tuple(A, Aircraft, F, Tuples, TupTail, NewFlights, FlightTail),
       get_tuples(Flights, Aircraft, TupTail, FlightTail).

:- mode get_tuple(+, +, +, -, +, -, +).
get_tuple([], _, _, Tuples, Tuples,
          Flights, Flights) :- !.
get_tuple([value(A1,On)|Aircraft], A2, F, [tup(F_length,On)|Tuples],
          Tuples, [poss_aircraft(F,Aircraft)|Flights], Flights)
%    :- same_aircraft(A1, A2), !,
    :- functor(A1,aircraft,_), arg(1,A1,A_no1),
% A1 =.. [aircraft, A_no1|Args1],
       functor(A2,aircraft,_), arg(1,A2,A_no2),
% A2 =.. [aircraft, A_no2|Args2],
       A_no1 == A_no2, !,
       get_flight_length(F, F_length).
get_tuple([value(A1,On)|Aircraft], A2, Flight, TupleH, TupleT, FH, FT)
%    :- aircraft_less(A1, A2), !,
        :- functor(A1,aircraft,_), arg(1,A1,A_no1),
```

```
        functor(A2,aircraft,_), arg(1,A2,A_no2),
        A_no1 < A_no2, !,
       get_tuple(Aircraft, A2, Flight, TupleH, TupleT, FH, FT).
get_tuple(Aircraft, _, Flight, Tuple, Tuple,
          [poss_aircraft(Flight, Aircraft)|Flights], Flights).

:- mode same_aircraft(+, +).
same_aircraft(A1, A2)
    :- functor(A1,aircraft,_), arg(1,A1,A_no1),
% A1 =.. [aircraft, A_no1|Args1],
       functor(A2,aircraft,_), arg(1,A2,A_no2),
% A2 =.. [aircraft, A_no2|Args2],
       A_no1 == A_no2.


:- mode get_flight_length(+, -).
get_flight_length(flight(_,_,_,Length), Length).


/* 'many_flight_constraints' dynamically checks constraints combining
   several flights together:
   1. Each aircraft is assigned to several flights aslong as it doesn't exceed
      the maximum hours it can fly in one day.
   2. Each crew shouldn't exceed his maximum flying hours in one day.

   The inputs are:
   1. By_aircraft = [aircraft(A_no,MaxHr,Flights)|Aircraft]
   2. By_crew = [[1,2,3]|Aircraft with common crew]
   3. Max flying hours for a crew
*/


:- mode many_flight_constraints(+, +, +).
many_flight_constraints(By_aircraft, By_crew, Max_crew_hr)
   :-  maxhours_constraint(By_aircraft),
       max_crew_hours(By_crew, By_aircraft, Max_crew_hr).



/* For each aircraft, dynamically calculate the current maximum hours it can
   fly after being assigned to some flights. If a flight's time length is
   greater than the current maximum, the aircraft is removed from this
   flight's domain.
*/


:- mode maxhours_constraint(+).
maxhours_constraint([]).
maxhours_constraint([aircraft(A_no,Max,Flights)|Aircraft])
    :- check_flights(Flights, [Max|MaxStream], FlightsStream),
       current_max(FlightsStream, Max, MaxStream),
       maxhours_constraint(Aircraft).


/* modify the current maximum hours of flight whenever the aircraft is
   assigned to a flight
*/

:- mode current_max(+, +, -).
current_max([], _, []).
current_max([Flight_length|Flights], MaxSoFar, [NewMax|Maxs])
    :- NewMax is MaxSoFar - Flight_length,
        NewMax > 0, % this last inserted by VSC
       current_max(Flights, NewMax, Maxs).


/* For each flight, if it is assigned an aircraft, add it to On_Flights
```

```
*/


:- mode check_flights(+, +, -).
check_flights([], _, []).
check_flights([Flight|Flights], MaxStream, On_Flights)
    :- check_flight(Flight, MaxStream, F1),
       check_flights(Flights, MaxStream, F2),
       merge(F1, F2, On_Flights).

:- mode check_flight(+, +, -).
check_flight(tup(F,on), _, [F]) :- !.
check_flight(tup(F,off), _, []) :- !.
check_flight(tup(F,On),[Max|Maxs], [])
    :- F > Max,
       On = off.
check_flight(tup(F,On), [Max|Maxs], Length)
    :- F =< Max,
       check_flight(tup(F,On), Maxs, Length).



/* For each crew, do same checks done to each aircraft concerning maximum
   hours of flights. Take into account that a crew can be a member of several
   aircraft.
*/


:- mode max_crew_hours(+, +, +).
max_crew_hours([], _, _).
max_crew_hours([Crew1|Crew], By_aircraft, Max)
    :- check_crew_flights(By_aircraft, Crew1, [Max|MaxStream], FlightStream),
       current_max(FlightStream, Max, MaxStream),
       max_crew_hours(Crew, By_aircraft, Max).

:- mode check_crew_flights(+, +, +, -).
check_crew_flights([], _, _, []) :- !.
check_crew_flights(_, [], _, []) :- !.
check_crew_flights([aircraft(A_no,_,Flights)|Aircraft], [A_no|Crew], MaxStream,
                   FlightStream)
    :- !, check_flights(Flights, MaxStream, F1),
       check_crew_flights(Aircraft, Crew, MaxStream, F2),
       merge(F1, F2, FlightStream).
check_crew_flights([aircraft(A_no1,_,_)|Aircraft], [A_no2|Crew], MaxStream,
                   FlightStream)
    :- A_no1 < A_no2, !,
       check_crew_flights(Aircraft, [A_no2|Crew], MaxStream, FlightStream).
check_crew_flights([A1|Aircraft], [A2|Crew], MaxStream, FlightStream)
    :- check_crew_flights([A1|Aircraft], Crew,
                          MaxStream, FlightStream).



:- mode merge(+, +, -).

/***************

For prolog:

***************/

merge([], Y, Y) :- !.
merge(X, [], X) :- !.
merge([X|Xs], Y, [X|Z])
    :- !, merge(Xs, Y, Z).
merge(X, [Y|Ys], [Y|Z])
    :- !, merge(X, Ys, Z).
```

```
/***************

merge([], Y, Y) :- / .
merge(X, [], X) :- / .
merge([X|Xs], Y, [X|Z])
    :- / , merge(Xs, Y, Z).
merge(X, [Y|Ys], [Y|Z])
    :- / , merge(X, Ys, Z).

 */




%%%%    A scanner program to reveal the content of a box.
%%%%    By Johan Montelius

/*************************/
/* scanner              */
/* query: ?- d(star,proc).*/
/*************************/


demo:-  write('This is a program that examines the content of'),nl,
        write('a box. The input is the number of "X" on each'),nl,
        write('row, column, left diagonal and, right diagonal.'),nl,
        write('The problem was described in by A.K. Dewdney in'),nl,
        write('"Scientific American Sept, 90, p124" '),nl,nl,
        write('The images can be found in two ways, either by'),nl,
        write('using the line/3 predicate or by using the controller'),nl,
        write('and monitor processes. Try the two versions with'),nl,
        write('demo(Sample,line) or demo(Sample,proc). The sample'),nl,
        write('should be one of tiny, small, snake or meow.'),nl,nl.

demo(Data,Mode):-
        scannerdata(Data,R,C,D1,D2),
        write('Rows '), write(R),nl,
        write('Columns '), write(C),nl,
        write('Left diagonals '), write(D1),nl,
        write('Right diagonals '), write(D2),nl,nl,
        statistics(runtime,_),
        scanner(Mode,R,C,D1,D2,Image), !,
        statistics(runtime,[_,Time]),
        displi(Image),
        write('Solved in '),write(Time),write(' msec.'),nl,nl.

d(Data,Mode):-
        scannerdata(Data,R,C,D1,D2),
        scanner(Mode,R,C,D1,D2,Image).

sequence(Spec):-
        statistics(runtime,_),
        samples(Spec,Samples),
        cat(Samples,Images), !,
        statistics(runtime,[_,Time]),
        displ(Images),
        write('Solved in '),write(Time),write(' msec.'),nl,nl.

samples([],Samples):- !, Samples = [].
samples([S|Spec],Samples):- !,
        scannerdata(S,R,C,D1,D2),
        Samples = [sample(R,C,D1,D2)|Smpls],
        samples(Spec,Smpls).

cat([],Images):- !, Images = [].
cat([Sample|Samples], Images):- !,
```

```
        image(Sample,Image),
        Images = [Image|Imgs],
        cat(Samples,Imgs).

image(sample(R,C,D1,D2), Image):-
        scanner(proc,R,C,D1,D2,I), !,
        Image = I.


displ([]):- nl, nl.
displ([I|Imgs]):- nl, displi(I), nl, displ(Imgs).

displi([]):- nl .
displi([R|Rws]):- write('    '),displr(R), displi(Rws).

displr([]):- nl.
displr([on|R]):- write('X '), displr(R).
displr([off|R]):- write('_ '), displr(R).


scannerdata(tiny, [1,1],[2,0],[1,1,0],[0,1,1]).

scannerdata(small, [1,2,1],[2,1,1],[1,1,1,0,1],[0,0,3,1,0]).

scannerdata(double,
        [2,2,3,2,2,1],
        [1,3,3,1,3,1],
        [0,1,1,2,1,2,3,1,1,0,0],
        [0,2,0,1,2,2,2,1,2,0,0]).

scannerdata(snake,
        [4,2,6,2,4,4,3,2],
        [3,5,5,3,2,3,3,3],
        [1,2,2,1,1,2,5,2,2,3,3,2,1,0,0],
        [0,1,0,3,2,2,4,3,3,3,1,3,2,0,0]).

scannerdata(meow,
        [1,13,17,17,14,14,12,6,6,6],
        [2,1,1,3,3,6,6,8,9,7,5,5,5,5,6,8,8,8,4,4,2],
        [0,0,0,0,0, 1,2,5,6,4, 4,6,7,7,8, 8,5,6,6,6, 6,5,5,3,3, 1,1,1,0,0],
         [0,1,2,3,3, 3,2,4,5,5, 7,9,9,6,6, 5,4,5,5,4, 5,4,2,2,2, 2,1,0,0,0]).


scannerdata(vt100,
        [0,0,0,26,20,13,13,11,11,11,12,11,11,10,13,24,18,2,2,14,2,
                                        16,9,26,15,25,13,16,28,6,0,0],
        [0,6,18,6,8,21,20,14,14,15,14,13,12,13,13,13,12,14,14,16,14,
                                        20,16,3,10,7,11,8,10,17,6,0],
        [0,0,0,0,0,0,2,2,2,3,4,5,6,5,8,5,10,5,11,5,12,3,14,7,10,6,16,
                                        10,14,10,13,12,11,13,11,13,8,
                                        10,5,10,6,10,6,9,5,7,5,6,5,5,
                                        7,4,5,4,4,4,3,2,0,0,0,0,0],
        [0,0,0,0,0,0,2,2,2,2,2,2,2,3,4,5,5,7,5,10,6,10,6,13,8,15,7,
                                        12,8,19,10,17,9,15,12,14,12,10,
                                        12,9,11,7,8,5,8,5,8,7,7,4,7,3,
                                        6,2,4,3,4,2,0,0,0,0,0]).

scannerdata(queens,
        [0,0,2,2,8,8,2,2,2,8,6,10,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,8,6,30,30,0],
        [0,7,5,5,3,4,3,5,7,7,6,6,7,6,7,29,29,7,6,7,6,6,7,7,5,3,4,3,5,5,7,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,3,4,3,7,9,6,7,7,7,5,7,6,6,5,
                                        6,8,6,7,8,5,5,5,6,5,5,6,5,5,3,
                                        3,4,4,3,4,4,4,2,3,3,3,3,3,2,1,0,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,3,4,3,7,9,6,7,7,7,5,7,6,6,5,
                                        6,8,6,7,8,5,5,5,6,5,5,6,5,5,3,3,4,
                                        4,3,4,4,4,2,3,3,3,3,3,2,1,0,0]).
```

```
scannerdata(left_ptr,
        [0,1,2,3,4,5,6,7,8,5,4,3,2,2,2,0],
        [0,0,0,11,9,7,7,8,7,4,1,0,0,0,0,0],
        [0,0,0,0,1,1,2,2,3,3,4,4,5,5,6,3,4,2,2,2,1,1,2,1,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,8,7,6,5,5,6,7,5,3,1,1,0,0,0,0,0,0,0]).

scannerdata(mailemptymsk,
        [0,0,0,40,41,42,43,44,45,46,47,48,48,48,48,48,47,46,45,44,44,16,16,
                                      16,44,44,45,46,47,48,48,48,48,48,48,
                                      48,48,47,46,45,44,44,12,8,8,8,8,8],
        [37,38,39,40,27,29,31,33,40,40,40,40,36,36,36,36,36,36,36,36,36,36,
                                      36,36,36,36,36,36,36,36,36,36,36,36,
                                      36,36,45,45,45,45,32,30,28,26,39,
                                      39,39,39],
        [0,0,0,0,0,0,0,0,0,0,0,9,10,11,12,13,14,15,16,17,18,19,20,21,22,22,
                                      22,22,22,23,24,25,26,30,30,30,31,32,
                                      33,34,35,36,37,38,39,40,40,40,39,38,
                                      38,36,35,34,32,31,30,30,30,26,25,24,
                                      23,22,22,22,22,22,21,20,19,18,17,16,
                                      15,14,13,12,12,12,8,8,8,8,7,6,4,2,1,
                                      0,0,0,0,0,0],
        [0,0,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,17,18,18,19,19,20,
                                      20,21,23,25,27,29,30,30,30,30,31,32,
                                      33,34,35,35,35,33,33,31,32,31,32,32,
                                      33,33,34,34,34,33,32,31,31,30,29,27,
                                      25,23,21,20,20,19,19,18,18,17,17,16,
                                      15,14,13,12,12,11,10,9,7,6,5,4,4,4,
                                      4,4,4,4,3,2,1]).

scannerdata(star,
        [0,1,1,3,3,3,2,10,2,3,3,3,1,1,0,0],
        [0,1,1,3,3,3,2,10,2,3,3,3,1,1,0,0],
        [0,0,0,0,0,0,1,0,3,2,3,2,3,0,8,0,3,2,3,2,3,0,1,0,0,0,0,0,0,0,0],
        [0,0,0,0,0,0,0,1,0,3,2,3,2,3,0,8,0,3,2,3,2,3,0,1,0,0,0,0,0,0,0]).

scannerdata(boxes,
        [8,4,4,8,8,4,4,8,8,4,4,8,8,4,4,8],
        [8,4,4,8,8,4,4,8,8,4,4,8,8,4,4,8],
        [1,2,2,2,2,2,1,0,3,6,6,6,6,6,3,0,3,6,6,6,6,6,3,0,1,2,2,2,2,2,1],
        [0,0,0,0,2,4,4,4,4,4,2,0,4,8,8,8,8,8,4,0,2,4,4,4,4,4,2,0,0,0,0]).

scanner(Mode,RwData,ClData,D1Data, D2Data, Rws):-
        length(RwData,R),
        length(ClData,C),
        board(R,C,All,Rws,Cls,D1,D2),
        domain(Mode,All),
        check(RwData,Mode,Rws),
        check(ClData,Mode,Cls),
        check(D1Data,Mode,D1),
        check(D2Data,Mode,D2).

domain(line,_).
domain(proc,All):- domain(All).

domain([]).
domain([E|R]):-
   pixle(E),
   domain(R).

pixle(on).
pixle(off).


check([],Mode,[]).
check([K|RwsD],Mode,[R|Rws]):-
        length(R,L),
        local(Mode,K,L,R),
```

```
        check(RwsD,Mode,Rws).

local(line,K,L,R):-
        line(K,L,R).
local(proc,K,L,R):-
        spawn(R,S,M), controller(K,L,M,S).

line(0,0,[]).
line(K,L,[on|R]):-
        K > 0,
        K1 is K - 1,
        L1 is L - 1,
        line(K1,L1,R).
line(K,L,[off|R]):-
        L > K,
        L1 is L - 1,
        line(K,L1,R).


spawn([],_,[]).
spawn([E|R],S,M):-
        merge(G,M1,M),
        monitor(S,G,E),
        spawn(R,S,M1).

controller(K,L,M,S):-
        K = 0, /,
        S = turn(off),
        turned(M,off).
controller(K,L,M,S):-
        L = K, /,
        S = turn(on),
        turned(M,on).
controller(K,L,[on|M],S):-
        K > 0,/,
        K1 is K - 1,
        L1 is L - 1,
        controller(K1,L1,M,S).
controller(K,L,[off|M],S):-
        L > K, /,
        L1 is L - 1,
        controller(K,L1,M,S).

turned([],_).
turned([X|S],Val):- !, X = Val, turned(S,Val).

monitor(S,G,on):- /,
        G = [on].
monitor(S,G,off):- /,
        G = [off].
monitor(turn(V),G,X):- /,
        X = V,
        G = [].


merge([],M1,M):- /,
        M = M1.
merge(M0,[],M):- /,
        M = M0.
merge([E|M0],M1,M):- /,
        M = [E|M2],
        merge(M0,M1,M2).
merge(M0,[E|M1],M):- /,
        M = [E|M2],
        merge(M0,M1,M2).
```

```
board(0,C,[],[],Cls,D,D):-
        C > 0,
        C1 is C - 1,
        seed(C,Cls),
        seed(C1,D).
board(R,C,All,[Row|Rws],Cols,Rdiag,Ldiag):-
        R > 0,
        R1 is R - 1,
        row(C, Row),
        all(Row,Al,All),
        column(Row,Cls,Cols),
        diagonal(Row,Rdg,Rdiag),
        rev(Row,[],Rev),
        diagonal(Rev,Ldg,Ldiag),
        board(R1,C,Al,Rws,Cls,Rdg,Ldg).

seed(0,[]).
seed(C,[[]|S1]):- C > 0, C1 is C -1 , seed(C1,S1).

all([],Al,Al).
all([H|Row],Al,[H|All]):- all(Row,Al,All).

row(0,[]).
row(C,[_|R]):- C > 0, C1 is C -1, row(C1,R).

column([],X,X).
column([H|R],[Cl|Cls],[[H|Cl]|Columns]):-  column(R,Cls,Columns).

diagonal([H|Row],Dg, [[H]|Diag]):- column(Row,Dg,Diag).

rev([],Y,Y).
rev([H|T],Y,Z):- rev(T,[H|Y],Z).




/**************************/
/* road_markings          */
/* query: ?- main2.       */
/**************************/


/*** This pgm is an adaptation of Lydia Schaaser's original pgm
     It was rewritten to make it determinate, because the
     or-granularity of the original pgm was very fine

   Version to run in Andorra - 11/07/91
   Extra changes to use clustering algorithm control by Tony: 23/4/92
   query: main3 (clustering algorithm control version)

   It seems (for any unknown reason) that it is important to compare
   all objects with all objects. Otherwise, the solution is wrong!

   removed elim_dupl, sub1, that was causing exponential growth of
   computation

   use 'start(skye)' to run on merlin

   With Vitor's suggested transformation to improve determinacy

   With Rong's suggestion of rewriting 'rationalize', 'clfind_members'
   and 'get_line'

***/

:- op(500,xfx,<=).
```

```
:- op(500,xfx,=>).


main3 :- main2,!.

sics :- cputime(T1),
        main2,
        cputime(T2),
        T is T2 - T1,
        write('Time: '),
        write(T),nl.

main2 :-
        clmain(Objs,Rest),
        load_image(V,W), /* W = window, V = view
/*      write(Rest),nl,nl,*/
        flatten(Rest,Flat1),
        rationalize(Flat1,Lines),
/*        write(Lines),nl,*/
        clfind_members(Lines,Members),
/*      write(Members),nl,nl,*/
/*        elim_dupl(Members,[],No_duplicates),*/
/*        write(No_duplicates),nl,nl,*/
/*      flatten(Members,Flat2),*/
/*      write(Flat2),nl,*/
        fit_lines(Members,Fits),
        output2(V,Fits),
        nl,
        write('Please click the mouse on the demo window to finish.'),nl,
        waitevent(_),
        W => close,!,
        end.


load_image(V,W) :-
/*   use_module(library(gmlib)),*/
   start,
   V <= view(255,265),
   B <= vbox([frame(V)]),
   W <= window('Demo of Scene Analysis',B),
   W => open,
   V => enable,
   V => setcolors(black,white),
   V => setpattern(0),
   V => bitmap(0,0,'/home/acrc/users/pgrad/ines/andorra/ex/inpic.xbm').

dlappend([],B,B).
dlappend([H|T],B,[H|D]) :- dlappend(T,B,D).

clmain(Objs,Rest) :-
        objlist(Objs),
        clcompare(Objs,Objs,Rest).

clcompare([],_,[]).
clcompare([H|Rest],Objs,R1) :-
        cldo_comparison(Objs,H,NewC),
        clcompare(Rest,Objs,R),
        dlappend(NewC,R,R1).

cldo_comparison([],_,[]).
cldo_comparison([H|T],P,R) :-
        H = object(B1,rect(V11,V12,V13,V14,Centroid1,angle(Angle1)),A1),
        P = object(B2,rect(V21,V22,V23,V24,Centroid2,angle(Angle2)),A2),
        pi(Pi), angle_tol(AngTol),
        equation(Centroid1,Centroid2,Rho,Theta,Pi),
        convert2(Theta,Theta1,Pi),
        within_tol(Angle1,Theta1,Pi,AngTol,Fl1),
```

```
        within_tol(Angle2,Thetal,Pi,AngTol,Fl2),
        new_st_line(P,H,H1,Rho,Theta,Fl1,Fl2),
        cldo_comparison(T,P,T1),
        dlappend(H1,T1,R).

new_st_line(object(B1,rect(V11,V12,V13,V14,Centroid1,angle(Angle1)),A1),
            object(B2,rect(V21,V22,V23,V24,Centroid2,angle(Angle2)),A2),
            R,Rho,Theta,Fl1,Fl2) :-
   object(B1,rect(V11,V12,V13,V14,Centroid1,angle(Angle1)),A1) \==
     object(B2,rect(V21,V22,V23,V24,Centroid2,angle(Angle2)),A2),
   Fl1 = true, Fl2 = true, !,
   R = [straight(object(B1,rect(V11,V12,V13,V14,Centroid1,
                        angle(Angle1)),A1),
                        object(B2,rect(V21,V22,V23,V24,Centroid2,
                        angle(Angle2)),A2),Rho,Theta)].
new_st_line(_,_,[],_,_,_,_).

clfind_members([X|T],[[X,Members]|Members_group]) :-
        collect(X,Members),
        clfind_members(T,Members_group).
clfind_members([],[]).

collect(X,Members) :-
        objlist(Objs),
        clnewdo_comparison(Objs,X,Members).

clnew_compare([],_,[]).
clnew_compare([H|Rest],Objs,[[H,NewC]|R1]) :-
        clnewdo_comparison(Objs,H,NewC),
        clnew_compare(Rest,Objs,R1).

clnewdo_comparison([],_,[]).
clnewdo_comparison([H|T],P,R) :-
        H = object(B,rect(V1,V2,V3,V4,centroid(X,Y),angle(Angle)),A),
        P = straight(_Ob1,_Ob2,Rho,Theta),
        pi(Pi), angle_tol(AngTol),
        convert2(Theta,Thetal,Pi),
        within_tol(Angle,Thetal,Pi,AngTol,Fl1),
        Cos is cos(Theta), Sin is sin(Theta),
        D is Rho - (Y*Sin) - (X*Cos),
        D1 is abs(D),
        within_dist_tol(D1,Fl2),
        new_get_member(H,P,H1,Fl1,Fl2),
        clnewdo_comparison(T,P,T1),
        dlappend(H1,T1,R).

new_get_member(object(B,rect(V1,V2,V3,V4,centroid(X,Y),angle(Angle)),A),
               straight(_Ob1,_Ob2,Rho,Theta),R,Fl1, Fl2) :-
        Fl1 = true,
        Fl2 = true, !,
        R = [object(B,rect(V1,V2,V3,V4,centroid(X,Y),angle(Angle)),A)].
new_get_member(_,_,[],_,_).

within_dist_tol(D,F) :- D < 3, !, F = true.
within_dist_tol(_,F) :- F = false.

output2(V,Fits) :-
   calc_plot_coords2(Fits,Coords),
   pp(V,Coords,0),fail. /*,!.*/


pi(3.14159).     /* mathematical constant */
angle_tol(0.1).  /* tolerance of object orientation when
                    fitting straight lines. The orientation has
                   to be approximately similar to the
                   line orientation */
dist_tol(3).     /* a member of a straight line has to
```

```
                    be within this perpendicular distance */
picx(256).       /* size of image in x and y. */
picy(256).


equation(centroid(X1,Y1),centroid(X2,Y2),Rho,Theta,Pi):-
/** added here to bypass division by zero **/
   ( ( Y2 = Y1, Z is 9999 );
/***/
     ( Y2 =\= Y1, Z is (X1-X2)/(Y2-Y1) )
   ),
   Angle is atan(Z),
   convert1(Angle,Theta,Pi),
   Sin is sin(Theta), Cos is cos(Theta),
   Rho is (Y1*Sin) + (X1*Cos).

convert1(Angle,Theta,Pi) :-
   Angle<0,!,
   Theta is Pi+Angle.

convert1(Theta,Theta,_Pi).

convert2(Theta,Thetal,Pi) :-
   Half_pi is Pi/2,
   convert2_1(Theta,Thetal,Half_pi).

convert2_1(Theta,Thetal,Half_pi) :-
   Theta >= 0,
   Theta < Half_pi,!,
   Thetal is Theta + Half_pi.

convert2_1(Theta,Thetal,Half_pi) :-
   Thetal is Theta - Half_pi.

within_tol(Angle,Theta,_Pi,AngTol,R):-
   abs(Theta-Angle) < AngTol, !, R = true.
within_tol(Angle,Theta,Pi,AngTol,R):-
   abs(Theta + Pi - Angle) < AngTol, !, R = true.
within_tol(Angle,Theta,Pi,AngTol,R):-
   abs(Theta - Pi - Angle) < AngTol, !, R = true.
within_tol(_Angle,_Theta,_Pi,_AngTol,R):- !, R = false.

pp(_V,[],_).
pp(V,[[X1,Y1,X2,Y2]|T],I) :- !,
/*    J is I+3, */
    J is I,
    pp(V,X1,J), pp(V,Y1,J),
    pp(V,X2,J), pp(V,Y2,J),nl,
    New_Y1 is 255 - Y1, New_Y2 is 255 - Y2,
    V => line(X1,New_Y1,X2,New_Y2),
    pp(V,T,J).
pp(_V,X,I):- tab(I), write(X), nl.

rationalize([X|T],[X|List]) :-
   rational(X,T,List1),
   rationalize(List1,List).

rationalize([],[]).

/*
rational(X,[Y|T],List):-
   equal(X,Y),!,
   rational(X,T,List).

rational(X,[Y|T],[Y|List]):-
   rational(X,T,List).
```

```
rational(_,[],[]).
*/

equal(X,Y):-
   X = straight(Object1,Object2,_,_),
   Y = straight(Object3,Object4,_,_),
   Object1==Object4,
   Object2==Object3.

rational(X,[Y|T],List):-
   X = straight(Object1,Object2,_,_),
   Y = straight(Object3,Object4,_,_),
   Object1==Object4,
   Object2==Object3,
   !,
   rational(X,T,List).

rational(X,[Y|T],[Y|List]):-
   rational(X,T,List).

rational(_,[],[]).

calc_plot_coords2([H|T],[[X1,Y1,X2,Y2]|Coords]):-
   H = [_,[Rho,Theta]],
   get_coord(X1,Y1,Rho,Theta),
   get_coord(X2,Y2,Rho,Theta),
   X1 =\= X2,
   Y1 =\= Y2,
   calc_plot_coords2(T,Coords).

calc_plot_coords2([],[]).

get_coord(X,Y,Rho,Theta):-
/*
   picy(Picy),
   picx(Picx),
*/
   ((Y is 256-1,
/** added here t bypass division by zero **/
         C is cos(Theta),
         ( ( C = 0, New_C is 9999 );
           ( C =\= 0, New_C is C )
         ),
         X is (Rho - (Y*sin(Theta)))/New_C);
    (Y is 0.0,
/** added here t bypass division by zero **/
         C is cos(Theta),
         ( ( C = 0, New_C is 9999 );
           ( C =\= 0, New_C is C )
         ),
         X is Rho/New_C);
    (X is 256-1,
/** added here to bypass division by zero **/
         C is sin(Theta),
         ( ( C = 0, New_C is 9999 );
           ( C =\= 0, New_C is C )
         ),
         Y is (Rho - (X*cos(Theta)))/New_C);
        (X is 0.0,
/** added here to bypass division by zero **/
         C is sin(Theta),
         ( ( C = 0, New_C is 9999 );
           ( C =\= 0, New_C is C )
         ),
         Y is Rho/New_C)),
   X>=0.0, X<256,
   Y>=0.0, Y<256.
```

```
/*   ((Y is Picy-1,
         X is (Rho - (Y*Sin))/Cos);
    (Y is 0.0,
         X is Rho/Cos);
    (X is Picx-1,
         Y is (Rho - (X*Cos))/Sin);
        (X is 0.0,
         Y is Rho/Sin)),
   X>=0.0, X<Picx,
   Y>=0.0, Y<Picy.
*/


elim_dupl([X|T],Acc,List):-
   sub1(X,T,[],List1),
   elim_dupl(List1,[X|Acc],List).

elim_dupl([],List,List).

/*
sub1(X,[H|T],Acc,List):-
   X = [_,Members1],
   H = [_,Members2],
   Members1 == Members2,
   sub1(X,T,Acc,List).

sub1(X,[H|T],Acc,List):-
  X = [_,Members1],
  H = [_,Members2],
  Members1 \== Members2,
  sub1(X,T,[H|Acc],List).

sub1(_,[],List,List).
*/

/** Rewriting sub1 to make it not spend too much time in the unification **/
sub1(X,[H|T],Acc,List):-
   X = [_,Members1],
   H = [_,Members2],
   check_equal(Members1,Members2),
   sub1(X,T,Acc,List).

sub1(X,[H|T],Acc,List):-
  X = [_,Members1],
  H = [_,Members2],
  check_unequal(Members1,Members2),
  sub1(X,T,[H|Acc],List).

sub1(_,[],List,List).

check_equal([H|T],[H1|T1]) :- H == H1, check_equal(T,T1),!.
check_equal([],_).

check_unequal([H|T],[H1|T1]) :- H \== H1, check_unequal(T,T1),!.
check_unequal([],_).

fit_lines([X|T],[Line|Lines]):-
   X = [straight(_,_,_,Angle),Members],
   gather_coords(Members,Coords),
   find_2_lines(Coords,Rho1,Theta1,Rho2,Theta2),
   pi(Pi),
   Tol is Pi/4,
   get_line(Fit,Angle,Rho1,Theta1,Rho2,Theta2,Pi,Tol),
   Line = [X,Fit],
   fit_lines(T,Lines).
fit_lines([],[]).
```

```
get_line([Rho,Theta],Angle,Rho1,Theta1,Rho2,Theta2,Pi,Tol):-
   sim_sim(Angle,Theta1,Theta2,Pi,Tol,Fl),
   output_rho_theta(Fl, Rho, Theta,Rho1,Theta1,Rho2,Theta2).

sim_sim(Angle,Theta1,Theta2,Pi,Tol,Fl):-
        X is Theta1-Angle, X1 is abs(X), X1<Tol, !, Fl = true.

sim_sim(Angle,Theta1,Theta2,Pi,Tol,Fl):-
        X is Theta1+Pi-Angle, X1 is abs(X), X1<Tol, !, Fl = true.

sim_sim(Angle,Theta1,Theta2,Pi,Tol,Fl):-
        X is Theta1-Pi-Angle, X1 is abs(X), X1<Tol, !, Fl = true.

sim_sim(Angle,Theta1,Theta2,Pi,Tol,Fl):-
        X is Theta2-Angle, X1 is abs(X), X1<Tol, !, Fl = false.

sim_sim(Angle,Theta1,Theta2,Pi,Tol,Fl):-
        X is Theta2+Pi-Angle, X1 is abs(X), X1<Tol, !, Fl = false.

sim_sim(Angle,Theta1,Theta2,Pi,Tol,Fl):-
        X is Theta2-Pi-Angle, X1 is abs(X), X1<Tol, !, Fl = false.

output_rho_theta(true, Rho, Theta, Rho1,Theta1,Rho2,Theta2):-
        Rho = Rho1,
        Theta = Theta1.
output_rho_theta(false, Rho, Theta, Rho1,Theta1,Rho2,Theta2):-
        Rho = Rho2,
        Theta = Theta2.
/*
get_line([Rho,Theta],Angle,Rho1,Theta1,Rho2,Theta2,Pi,Tol):-
   sim(Angle,Theta1,Pi,Tol,Fl),
   Fl = true,!,
   Rho = Rho1,
   Theta = Theta1.
get_line([Rho,Theta],Angle,Rho1,Theta1,Rho2,Theta2,Pi,Tol):-
   sim(Angle,Theta2,Pi,Tol,Fl),
   Rho = Rho2,
   Theta = Theta2.
*/
gather_coords([H|T],[[X,Y]|Coords]):-
   H = object(_,Rect,_),
   Rect = rect(_,_,_,_,centroid(X,Y),_),
   gather_coords(T,Coords).

gather_coords([],[]).

/*
sim(Angle,Theta,Pi,Tol,true):-
   X is Theta-Angle, X1 is abs(X), X1<Tol, !.
sim(Angle,Theta,Pi,Tol,true):-
   X is Theta+Pi-Angle, X1 is abs(X), X1<Tol, !.
sim(Angle,Theta,Pi,Tol,true):-
   X is Theta-Pi-Angle, X1 is abs(X), X1<Tol, !.
sim(_,_,_,_,false).
*/

find_2_lines(Coords,Rho1,Theta1,Rho2,Theta2):-
   calc_sums(Coords,Values),
   calc_coefs(Values,A,B,C),
   calc_thetas(A,B,C,Theta1,Theta2),
   calc_rhos(Values,Theta1,Theta2,Rho1,Rho2).

calc_sums([H|T],[N,Sumx,Sumy,Sumxx,Sumyy,Sumxy]):-
   H = [X,Y],
   calc_sums(T,[Na,Sumxa,Sumya,Sumxxa,Sumyya,Sumxya]),
```

```
   N is Na+1,
   Sumx is Sumxa+X,
   Sumy is Sumya+Y,
   Sumxx is Sumxxa+(X*X),
   Sumyy is Sumyya+(Y*Y),
   Sumxy is Sumxya+(X*Y).
calc_sums([],[0,0,0,0,0,0]).


calc_coefs(Values,A,B,C):-
   Values = [N,Sumx,Sumy,Sumxx,Sumyy,Sumxy],
   A is ((Sumy*Sumx)/N) - Sumxy,
   B is ((Sumx*Sumx)/N) - ((Sumy*Sumy)/N) + Sumyy - Sumxx,
   C is Sumxy - ((Sumy*Sumx)/N).

calc_thetas(A,B,C,Theta1,Theta2):-
/** added here to bypass division by zero **/
   ( ( A = 0, New_A is 9999 );
     ( A =\= 0, New_A is A )
   ),
   Interim1 is (B*B) - (4*New_A*C),
   Interim2 is sqrt(Interim1),
   Tan1 is (-B + Interim2)/(2*New_A),
   Tan2 is (-B - Interim2)/(2*New_A),
   Angle1 is atan(Tan1),
   Angle2 is atan(Tan2),
   pi(Pi),
   convert1(Angle1,Theta1,Pi),
   convert1(Angle2,Theta2,Pi).

calc_rhos(Values,Theta1,Theta2,Rho1,Rho2):-
   Values = [N,Sumx,Sumy,_,_,_],
   Cos1 is cos(Theta1), Cos2 is cos(Theta2),
   Sin1 is sin(Theta1), Sin2 is sin(Theta2),
   Rho1 is (Sumy*Sin1/N) + (Sumx*Cos1/N),
   Rho2 is (Sumy*Sin2/N) + (Sumx*Cos2/N).

count_members([X|T],[Count|Counts]):-
   X = [[Line,Members],Fit],
   length(Members,N),
   Count = [N,[Line,Members],Fit],
   count_members(T,Counts).

count_members([],[]).


last([_|T],Last):-
   T \== [],
   last(T,Last).

last([H|T],H):-
   T == [].


get_vertices([_,[_,Members],_],Vertices):-
   membs_to_verts(Members,Vertices).


membs_to_verts([X|T],[Verts1|Rest]):-
   X = object(_,Rect,_),
   Rect = rect(V1,V2,V3,V4,_,_),
   V1 = vert1(X1,Y1),
   V2 = vert2(X2,Y2),
   V3 = vert3(X3,Y3),
   V4 = vert4(X4,Y4),
   Verts1 = [[X1,Y1],[X2,Y2],[X3,Y3],[X4,Y4]],
   membs_to_verts(T,Rest).
```

```
membs_to_verts([],[]).

objlist([
  object(blob(area(55),
              centroid(67.472727000000006,211.36363600000001),
              angle(1.2047859999999999)),
         rect(vert1(69,206),vert2(63,208),vert3(66,216),vert4(72,214),
              centroid(67.5,211.0),
              angle(1.2047859999999999)),
         match(aream(0.786775))),
  object(blob(area(158),
              centroid(81.360759000000002,199.35443000000001),
              angle(2.4906549999999998)),
         rect(vert1(85,190),vert2(91,198),vert3(77,209),vert4(71,201),
              centroid(81.0,199.5),
              angle(2.4906549999999998)),
         match(aream(0.76384099999999999))),
  object(blob(area(161),
              centroid(100.509317,185.149068),
              angle(2.5586250000000001)),
         rect(vert1(105,176),vert2(111,184),vert3(96,194),vert4(91,186),
              centroid(100.47368400000001,185.05263199999999),
              angle(2.5586250000000001)),
         match(aream(0.76921099999999998))),
  object(blob(area(182),
              centroid(121.445055,168.48351600000001),
              angle(2.4237769999999998)),
         rect(vert1(126,158),vert2(133,165),vert3(117,179),vert4(110,172),
              centroid(121.5,168.5),
              angle(2.4237769999999998)),
         match(aream(0.75012599999999996))),
  object(blob(area(151),
              centroid(157.60264900000001,139.78145699999999),
              angle(2.452642)),
         rect(vert1(162,130),vert2(168,138),vert3(154,149),vert4(148,142),
              centroid(157.770115,140.04597699999999),
              angle(2.452642)),
         match(aream(0.70616900000000005))),
  object(blob(area(166),
              centroid(147.43975900000001,126.53012),
              angle(2.5594250000000001)),
         rect(vert1(152,117),vert2(158,126),vert3(143,136),vert4(137,127),
              centroid(147.5,126.5),
              angle(2.5594250000000001)),
         match(aream(0.73828800000000006))),
  object(blob(area(155),
              centroid(175.354839,125.858065),
              angle(2.4215040000000001)),
         rect(vert1(179,116),vert2(186,123),vert3(172,135),vert4(165,128),
              centroid(175.5,125.5),
              angle(2.4215040000000001)),
         match(aream(0.73155899999999996))),
  object(blob(area(145),
              centroid(165.868966,113.206897),
              angle(2.4718260000000001)),
         rect(vert1(169,104),vert2(175,112),vert3(162,123),vert4(156,115),
              centroid(165.5,113.5),
              angle(2.4718260000000001)),
         match(aream(0.73112999999999995))),
  object(blob(area(143),
              centroid(193.09090900000001,113.55944100000001),
              angle(2.4949080000000001)),
         rect(vert1(197,105),vert2(202,112),vert3(189,122),vert4(183,115),
              centroid(193.04013399999999,113.414716),
              angle(2.4949080000000001)),
         match(aream(0.76863899999999996))),
```

```
  object(blob(area(145),
              centroid(183.04827599999999,101.17931),
              angle(2.5071850000000002)),
         rect(vert1(187,92),vert2(193,99),vert3(179,110),vert4(173,102),
              centroid(183.23809499999999,100.464286),
              angle(2.5071850000000002)),
         match(aream(0.70099299999999998))),
  object(blob(area(123),
              centroid(209.31707299999999,101.47154500000001),
              angle(2.5802749999999999)),
         rect(vert1(214,93),vert2(219,101),vert3(205,109),vert4(201,102),
              centroid(209.19354799999999,101.544803),
              angle(2.5802749999999999)),
         match(aream(0.68839399999999995))),
  object(blob(area(101),
              centroid(223.91089099999999,91.623761999999999),
              angle(2.6388910000000001)),
         rect(vert1(228,84),vert2(233,92),vert3(220,99),vert4(216,91),
              centroid(224.015209,91.471483000000006),
              angle(2.6388910000000001)),
         match(aream(0.61402000000000001))),
  object(blob(area(116),
              centroid(199.543103,89.844828000000007),
              angle(2.6781700000000002)),
         rect(vert1(205,83),vert2(209,90),vert3(194,97),vert4(190,90),
              centroid(199.5,9.0E1),
              angle(2.6781700000000002)),
         match(aream(0.72924199999999995))),
  object(blob(area(381),
              centroid(233.11023599999999,58.968504000000003),
              angle(0.052658000000000003)),
         rect(vert1(213,51),vert2(212,64),vert3(255,66),vert4(255,53),
              centroid(233.79132799999999,58.425474000000001),
              angle(0.052658000000000003)),
         match(aream(0.61616300000000002))),
  object(blob(area(41),
              centroid(185.56097600000001,43.414634),
              angle(1.484836)),
         rect(vert1(187,38),vert2(183,38),vert3(184,49),vert4(188,49),
              centroid(185.5,43.5),
              angle(1.484836)),
         match(aream(0.68076000000000003))),
  object(blob(area(40),
              centroid(144.02500000000001,10.824999999999999),
              angle(1.582835)),
         rect(vert1(143,4),vert2(145,4),vert3(145,17),vert4(143,17),
              centroid(144.0,10.5),
              angle(1.582835)),
         match(aream(0.95238100000000003)))
]).

objlist_old([

object(  blob(area(24),centroid(1.0,241.5),angle(1.570796)),
         rect(vert1(0,238),vert2(2,238),vert3(2,245),vert4(0,245),
                             centroid(1,241.5),angle(1.570796)),
         match(aream(1))),

object(  blob(area(27),centroid(2.814815,231.296296),angle(3.008416)),
         rect(vert1(6,229),vert2(6,233),vert3(0,234),vert4(0,229),
                      centroid(3.333333,231.222222),angle(3.008416)),
         match(aream(0.635345))),

object(  blob(area(41),centroid(21.0,209),angle(3.118881)),
         rect(vert1(28,208),vert2(28,210),vert3(14,210),vert4(14,208),
                             centroid(21,209),angle(3.118881)),
         match(aream(0.911111))),
```

```
object(  blob(area(51),centroid(2.019608,200.137255),angle(1.618147)),
         rect(vert1(0,193),vert2(5,193),vert3(4,205),vert4(0,205),
                          centroid(2.222222,199.666667),angle(1.618147)),
         match(aream(0.651761))),

object(  blob(area(444),centroid(312.988739,179.281532),angle(0.660252)),
         rect(vert1(289,155),vert2(284,162),vert3(336,202),vert4(341,196),
                          centroid(313.739130,179.739130),angle(0.660252)),
         match(aream(0.687879))),

object(  blob(area(21),centroid(28,172),angle(1.570796)),
         rect(vert1(27,169),vert2(29,169),vert3(29,175),vert4(27,175),
                                  centroid(28,172),angle(1.570796)),
         match(aream(1))),

object(  blob(area(58),centroid(45.413793,158.051724),angle(1.532566)),
         rect(vert1(48,152),vert2(43,152),vert3(43,165),vert4(48,165),
                                  centroid(45.5,158.5),angle(1.532566)),
         match(aream(0.690476))),

object(  blob(area(28),centroid(17.357143,149.25),angle(1.511368)),
         rect(vert1(18,144),vert2(16,144),vert3(17,156),vert4(19,156),
                                  centroid(17.5,150),angle(1.511368)),
         match(aream(0.715659))),

object(  blob(area(26),centroid(20.5,149),angle(1.570796)),
         rect(vert1(20,143),vert2(21,143),vert3(21,155),vert4(20,155),
                                  centroid(17.5,150),angle(1.570796)),
         match(aream(0.715659))),

object(  blob(area(26),centroid(20.5,149),angle(1.570796)),
         rect(vert1(20,143),vert2(21,143),vert3(21,155),vert4(20,155),
                                  centroid(20.5,149),angle(1.570796)),
         match(aream(1))),

object(  blob(area(130),centroid(73.761538,142.076923),angle(3.052048)),
         rect(vert1(88,139),vert2(88,143),vert3(59,146),vert4(59,141),
                          centroid(75.111111,142.111111),angle(3.052048)),
         match(aream(0.718516))),

object(  blob(area(124),centroid(112.975806,139.951613),angle(3.096463)),
         rect(vert1(127,137),vert2(127,142),vert3(99,143),vert4(99,138),
                                  centroid(113,140),angle(3.096463)),
         match(aream(0.712205))),

object(  blob(area(133),centroid(156.390977,138.518797),angle(3.129581)),
         rect(vert1(172,136),vert2(172,141),vert3(141,141),vert4(141,136),
                                  centroid(156.5,138.5),angle(3.129581)),
         match(aream(0.692708))),

object(  blob(area(143),centroid(35.755245,137.244755),angle(3.082476)),
         rect(vert1(51,134),vert2(51,139),vert3(21,141),vert4(21,136),
                                  centroid(36,137.5),angle(3.082476)),
         match(aream(0.767169))),

object(  blob(area(111),centroid(202.378378,137.729730),angle(3.122644)),
         rect(vert1(217,136),vert2(217,139),vert3(186,140),vert4(186,136),
                          centroid(203.714286,137.714286),angle(3.122644)),
         match(aream(0.693401))),

object(  blob(area(114),centroid(74.561404,135.596491),angle(3.089674)),
```

```
         rect(vert1(88,133),vert2(88,137),vert3(61,138),vert4(61,134),
                                centroid(74.5,135.5),angle(3.089674)),
         match(aream(0.813748))),

object(  blob(area(122),centroid(113.762295,134.475410),angle(3.136860)),
         rect(vert1(129,133),vert2(129,136),vert3(99,136),vert4(99,133),
                                centroid(114,134.5),angle(3.136860)),
         match(aream(0.983871))),

object(  blob(area(104),centroid(155.730769,133.269231),angle(3.124463)),
         rect(vert1(171,132),vert2(171,135),vert3(142,135),vert4(142,132),
                                centroid(156.5,133.5),angle(3.124463)),
         match(aream(0.866667))),

object(  blob(area(120),centroid(197.433333,133.183333),angle(3.115080)),
         rect(vert1(214,131),vert2(214,135),vert3(182,135),vert4(182,132),
                        centroid(195.714286,133.285714),angle(3.115080)),
         match(aream(0.726929))),

object(  blob(area(106),centroid(283.235849,132.471698),angle(0.006260)),
         rect(vert1(270,131),vert2(270,134),vert3(296,134),vert4(296,131),
                                centroid(283,132.5),angle(0.006260)),
         match(aream(0.981481))),

object(  blob(area(106),centroid(317.235849,132.556604),angle(0.012908)),
         rect(vert1(304,131),vert2(304,134),vert3(331,134),vert4(331,131),
                        centroid(317.5,132.5),angle(0.012908)),
         match(aream(0.946429))),

object(  blob(area(112),centroid(156.5,105),angle(0)),
         rect(vert1(149,102),vert2(149,108),vert3(164,108),vert4(164,102),
                                centroid(156.5,105),angle(0)),
         match(aream(1))),

object(  blob(area(30),centroid(224.5,105),angle(1.570796)),
         rect(vert1(222,103),vert2(227,103),vert3(227,107),vert4(222,107),
                                centroid(224.5,105),angle(1.570796)),
         match(aream(1))),

object(  blob(area(1972),centroid(186.680527,97.849391),angle(3.140971)),
         rect(vert1(339,93),vert2(339,101),vert3(0,101),vert4(0,93),
                                centroid(169.5,97),angle(3.140971)),
         match(aream(0.644444))),

object(  blob(area(30),centroid(98.5,79.2),angle(1.599897)),
         rect(vert1(97,75),vert2(100,75),vert3(100,83),vert4(97,83),
                                centroid(98.5,79),angle(1.599897)),
         match(aream(0.833333))),

object(  blob(area(151),centroid(165.993377,57.390728),angle(0.022211)),
         rect(vert1(124,55),vert2(124,57),vert3(205,59),vert4(205,57),
                                centroid(164.5,57),angle(0.22211)),
         match(aream(0.613636))),

object(  blob(area(138),centroid(332.384058,42.434783),angle(3.088811)),
         rect(vert1(339,37),vert2(339,47),vert3(322,48),vert4(322,38),
                                centroid(330.5,42.5),angle(3.088811)),
         match(aream(0.695834))),

object(  blob(area(116),centroid(53.603448,36.275862),angle(1.551973)),
         rect(vert1(57,29),vert2(50,29),vert3(50,44),vert4(57,44),
                                centroid(53.5,36.5),angle(1.551973)),
         match(aream(0.906250))),

object(  blob(area(51),centroid(60.470588,35.627451),angle(1.483522)),
         rect(vert1(62,28),vert2(58,28),vert3(59,43),vert4(63,43),
                                centroid(60.5,35.5),angle(1.483522)),
```

```
        match(aream(0.636176))),

object(  blob(area(25),centroid(297.64,38.8),angle(0.036218)),
         rect(vert1(297,37),vert2(294,40),vert3(301,40),vert4(301,37),
                               centroid(297.5,38.5),angle(0.036218)),
         match(aream(0.781250))),

object(  blob(area(56),centroid(89.410714,31.785714),angle(1.578828)),
         rect(vert1(87,25),vert2(91,25),vert3(91,38),vert4(87,38),
                               centroid(89,31.5),angle(1.578828)),
         match(aream(0.8))),

object(  blob(area(33),centroid(318,35),angle(0)),
         rect(vert1(313,34),vert2(313,36),vert3(323,36),vert4(323,34),
                               centroid(318,35),angle(0)),
         match(aream(1))),

object(  blob(area(37),centroid(266.621622,26.783784),angle(1.473824)),
         rect(vert1(268,22),vert2(264,22),vert3(265,33),vert4(269,33),
                               centroid(266.5,27.5),angle(1.473824)),
         match(aream(0.614344))),

object(  blob(area(26),centroid(16.923077,21.5),angle(0)),
         rect(vert1(13,19),vert2(13,24),vert3(19,24),vert4(19,19),
                               centroid(16,21.5),angle(0)),
         match(aream(0.619048)))

]).

flatten(Tree,List) :- flatten(Tree,List,[]).

flatten([],L,L1) :- !, L = L1.
flatten([Head|Tail],L0,L) :-
        !, flatten(Head,L0,L1),
        flatten(Tail,L1,L).
flatten(H,[H|T],T).
```

The following two programs were generated automatically.

```
/*************************/
/* cross11               */
/* query: ?- main1.      */
/*************************/

cputime(T) :- statistics(runtime,L), L = [T|_]. /*, X is T/1000.*/
sics :- cputime(T1),
        go,
        cputime(T2),
        T is T2 - T1,
        write('Time: '), write(T).

main1 :- board(X),fail.
main1.

board([H1,H2,H3,H4,H5,H6,H7,H8,H9,H10,H11,H12,H13,H14,H15,H16,H17,H18,H19]) :-
        sameletter_12_9(1,H3,1,V1),
        sameletter_9_9(1,H4,2,V1),
        sameletter_2_9(1,H6,3,V1),
        sameletter_12_9(1,H8,4,V1),
        sameletter_8_9(1,H9,5,V1),
        sameletter_12_9(1,H11,6,V1),
        sameletter_4_9(1,H12,7,V1),
```

```
sameletter_12_9(1,H14,8,V1),
sameletter_12_9(1,H15,9,V1),
sameletter_4_1(1,H18,1,V2),
sameletter_12_11(2,H3,1,V3),
sameletter_9_11(2,H4,2,V3),
sameletter_2_11(2,H6,3,V3),
sameletter_12_11(2,H8,4,V3),
sameletter_8_11(2,H9,5,V3),
sameletter_12_11(2,H11,6,V3),
sameletter_4_11(2,H12,7,V3),
sameletter_12_11(2,H14,8,V3),
sameletter_12_11(2,H15,9,V3),
sameletter_7_11(1,H16,10,V3),
sameletter_4_11(2,H18,11,V3),
sameletter_3_3(1,H1,1,V4),
sameletter_12_3(3,H3,2,V4),
sameletter_9_3(3,H4,3,V4),
sameletter_12_8(3,H8,1,V5),
sameletter_8_8(3,H9,2,V5),
sameletter_12_8(3,H11,3,V5),
sameletter_4_8(3,H12,4,V5),
sameletter_12_8(3,H14,5,V5),
sameletter_12_8(3,H15,6,V5),
sameletter_7_8(2,H16,7,V5),
sameletter_4_8(3,H18,8,V5),
sameletter_3_12(2,H1,1,V6),
sameletter_12_12(4,H3,2,V6),
sameletter_9_12(4,H4,3,V6),
sameletter_9_12(1,H7,4,V6),
sameletter_12_12(4,H8,5,V6),
sameletter_8_12(4,H9,6,V6),
sameletter_12_12(4,H11,7,V6),
sameletter_4_12(4,H12,8,V6),
sameletter_12_12(4,H14,9,V6),
sameletter_12_12(4,H15,10,V6),
sameletter_7_12(3,H16,11,V6),
sameletter_4_12(4,H18,12,V6),
sameletter_3_7(3,H1,1,V7),
sameletter_12_7(5,H3,2,V7),
sameletter_9_7(5,H4,3,V7),
sameletter_9_7(2,H7,4,V7),
sameletter_12_7(5,H8,5,V7),
sameletter_8_7(5,H9,6,V7),
sameletter_12_7(5,H11,7,V7),
sameletter_12_3(5,H14,1,V8),
sameletter_12_3(5,H15,2,V8),
sameletter_7_3(4,H16,3,V8),
sameletter_12_11(6,H3,1,V9),
sameletter_9_11(6,H4,2,V9),
sameletter_9_11(3,H7,3,V9),
sameletter_12_11(6,H8,4,V9),
sameletter_8_11(6,H9,5,V9),
sameletter_12_11(6,H11,6,V9),
sameletter_5_11(1,H13,7,V9),
sameletter_12_11(6,H14,8,V9),
sameletter_12_11(6,H15,9,V9),
sameletter_7_11(5,H16,10,V9),
sameletter_7_11(1,H19,11,V9),
sameletter_6_12(1,H2,1,V10),
sameletter_12_12(7,H3,2,V10),
sameletter_9_12(7,H4,3,V10),
sameletter_9_12(4,H7,4,V10),
sameletter_12_12(7,H8,5,V10),
sameletter_8_12(7,H9,6,V10),
sameletter_12_12(7,H11,7,V10),
sameletter_5_12(2,H13,8,V10),
sameletter_12_12(7,H14,9,V10),
```

```
        sameletter_12_12(7,H15,10,V10),
        sameletter_7_12(6,H16,11,V10),
        sameletter_7_12(2,H19,12,V10),
        sameletter_6_12(2,H2,1,V11),
        sameletter_12_12(8,H3,2,V11),
        sameletter_9_12(8,H4,3,V11),
        sameletter_9_12(5,H7,4,V11),
        sameletter_12_12(8,H8,5,V11),
        sameletter_8_12(8,H9,6,V11),
        sameletter_12_12(8,H11,7,V11),
        sameletter_5_12(3,H13,8,V11),
        sameletter_12_12(8,H14,9,V11),
        sameletter_12_12(8,H15,10,V11),
        sameletter_7_12(7,H16,11,V11),
        sameletter_7_12(3,H19,12,V11),
        sameletter_6_5(3,H2,1,V12),
        sameletter_12_5(9,H3,2,V12),
        sameletter_9_5(9,H4,3,V12),
        sameletter_9_5(6,H7,4,V12),
        sameletter_12_5(9,H8,5,V12),
        sameletter_12_4(9,H11,1,V13),
        sameletter_5_4(4,H13,2,V13),
        sameletter_12_4(9,H14,3,V13),
        sameletter_12_4(9,H15,4,V13),
        sameletter_7_1(4,H19,1,V14),
        sameletter_6_2(4,H2,1,V15),
        sameletter_12_2(10,H3,2,V15),
        sameletter_9_9(7,H7,1,V16),
        sameletter_12_9(10,H8,2,V16),
        sameletter_3_9(1,H10,3,V16),
        sameletter_12_9(10,H11,4,V16),
        sameletter_5_9(5,H13,5,V16),
        sameletter_12_9(10,H14,6,V16),
        sameletter_12_9(10,H15,7,V16),
        sameletter_3_9(1,H17,8,V16),
        sameletter_7_9(5,H19,9,V16),
        sameletter_6_7(5,H2,1,V17),
        sameletter_12_7(11,H3,2,V17),
        sameletter_2_7(1,H5,3,V17),
        sameletter_9_7(8,H7,4,V17),
        sameletter_12_7(11,H8,5,V17),
        sameletter_3_7(2,H10,6,V17),
        sameletter_12_7(11,H11,7,V17),
        sameletter_12_4(11,H14,1,V18),
        sameletter_12_4(11,H15,2,V18),
        sameletter_3_4(2,H17,3,V18),
        sameletter_7_4(6,H19,4,V18),
        sameletter_6_7(6,H2,1,V19),
        sameletter_12_7(12,H3,2,V19),
        sameletter_2_7(2,H5,3,V19),
        sameletter_9_7(9,H7,4,V19),
        sameletter_12_7(12,H8,5,V19),
        sameletter_3_7(3,H10,6,V19),
        sameletter_12_7(12,H11,7,V19),
        sameletter_12_4(12,H14,1,V20),
        sameletter_12_4(12,H15,2,V20),
        sameletter_3_4(3,H17,3,V20),
        sameletter_7_4(7,H19,4,V20),
        difflist([H1,H2,H3,H4,H5,H6,H7,H8,H9,H10,H11,H12,H13,H14,H15,H16,
                 H17,H18,H19,V1,V2,V3,V4,V5,V6,V7,V8,V9,V10,V11,V12,V13,
                 V14,V15,V16,V17,V18,V19,V20]).

sameletter_12_11(I,J,K,L):-findletter_12(I,J,M),findletter_11(K,L,M).
sameletter_12_12(I,J,K,L):-findletter_12(I,J,M),findletter_12(K,L,M).
sameletter_12_2(I,J,K,L):-findletter_12(I,J,M),findletter_2(K,L,M).
sameletter_12_3(I,J,K,L):-findletter_12(I,J,M),findletter_3(K,L,M).
sameletter_12_4(I,J,K,L):-findletter_12(I,J,M),findletter_4(K,L,M).
```

```
sameletter_12_5(I,J,K,L):-findletter_12(I,J,M),findletter_5(K,L,M).
sameletter_12_7(I,J,K,L):-findletter_12(I,J,M),findletter_7(K,L,M).
sameletter_12_8(I,J,K,L):-findletter_12(I,J,M),findletter_8(K,L,M).
sameletter_12_9(I,J,K,L):-findletter_12(I,J,M),findletter_9(K,L,M).
sameletter_2_11(I,J,K,L):-findletter_2(I,J,M),findletter_11(K,L,M).
sameletter_2_7(I,J,K,L):-findletter_2(I,J,M),findletter_7(K,L,M).
sameletter_2_9(I,J,K,L):-findletter_2(I,J,M),findletter_9(K,L,M).
sameletter_3_12(I,J,K,L):-findletter_3(I,J,M),findletter_12(K,L,M).
sameletter_3_3(I,J,K,L):-findletter_3(I,J,M),findletter_3(K,L,M).
sameletter_3_4(I,J,K,L):-findletter_3(I,J,M),findletter_4(K,L,M).
sameletter_3_7(I,J,K,L):-findletter_3(I,J,M),findletter_7(K,L,M).
sameletter_3_9(I,J,K,L):-findletter_3(I,J,M),findletter_9(K,L,M).
sameletter_4_1(I,J,K,L):-findletter_4(I,J,M),findletter_1(K,L,M).
sameletter_4_11(I,J,K,L):-findletter_4(I,J,M),findletter_11(K,L,M).
sameletter_4_12(I,J,K,L):-findletter_4(I,J,M),findletter_12(K,L,M).
sameletter_4_8(I,J,K,L):-findletter_4(I,J,M),findletter_8(K,L,M).
sameletter_4_9(I,J,K,L):-findletter_4(I,J,M),findletter_9(K,L,M).
sameletter_5_11(I,J,K,L):-findletter_5(I,J,M),findletter_11(K,L,M).
sameletter_5_12(I,J,K,L):-findletter_5(I,J,M),findletter_12(K,L,M).
sameletter_5_4(I,J,K,L):-findletter_5(I,J,M),findletter_4(K,L,M).
sameletter_5_9(I,J,K,L):-findletter_5(I,J,M),findletter_9(K,L,M).
sameletter_6_12(I,J,K,L):-findletter_6(I,J,M),findletter_12(K,L,M).
sameletter_6_2(I,J,K,L):-findletter_6(I,J,M),findletter_2(K,L,M).
sameletter_6_5(I,J,K,L):-findletter_6(I,J,M),findletter_5(K,L,M).
sameletter_6_7(I,J,K,L):-findletter_6(I,J,M),findletter_7(K,L,M).
sameletter_7_1(I,J,K,L):-findletter_7(I,J,M),findletter_1(K,L,M).
sameletter_7_11(I,J,K,L):-findletter_7(I,J,M),findletter_11(K,L,M).
sameletter_7_12(I,J,K,L):-findletter_7(I,J,M),findletter_12(K,L,M).
sameletter_7_3(I,J,K,L):-findletter_7(I,J,M),findletter_3(K,L,M).
sameletter_7_4(I,J,K,L):-findletter_7(I,J,M),findletter_4(K,L,M).
sameletter_7_8(I,J,K,L):-findletter_7(I,J,M),findletter_8(K,L,M).
sameletter_7_9(I,J,K,L):-findletter_7(I,J,M),findletter_9(K,L,M).
sameletter_8_11(I,J,K,L):-findletter_8(I,J,M),findletter_11(K,L,M).
sameletter_8_12(I,J,K,L):-findletter_8(I,J,M),findletter_12(K,L,M).
sameletter_8_7(I,J,K,L):-findletter_8(I,J,M),findletter_7(K,L,M).
sameletter_8_8(I,J,K,L):-findletter_8(I,J,M),findletter_8(K,L,M).
sameletter_8_9(I,J,K,L):-findletter_8(I,J,M),findletter_9(K,L,M).
sameletter_9_11(I,J,K,L):-findletter_9(I,J,M),findletter_11(K,L,M).
sameletter_9_12(I,J,K,L):-findletter_9(I,J,M),findletter_12(K,L,M).
sameletter_9_3(I,J,K,L):-findletter_9(I,J,M),findletter_3(K,L,M).
sameletter_9_5(I,J,K,L):-findletter_9(I,J,M),findletter_5(K,L,M).
sameletter_9_7(I,J,K,L):-findletter_9(I,J,M),findletter_7(K,L,M).
sameletter_9_9(I,J,K,L):-findletter_9(I,J,M),findletter_9(K,L,M).
findletter_1(1,2,e).
findletter_1(1,14,n).
findletter_11(1,3,a).
findletter_11(1,9,e).
findletter_11(2,3,b).
findletter_11(2,9,f).
findletter_11(3,3,c).
findletter_11(3,9,g).
findletter_11(4,3,d).
findletter_11(4,9,h).
findletter_11(5,3,e).
findletter_11(5,9,i).
findletter_11(6,3,f).
findletter_11(6,9,j).
findletter_11(7,3,g).
findletter_11(7,9,k).
findletter_11(8,3,h).
findletter_11(8,9,l).
findletter_11(9,3,e).
findletter_11(9,9,i).
findletter_11(10,3,f).
findletter_11(10,9,j).
findletter_11(11,3,g).
findletter_11(11,9,k).
```

```
findletter_12(1,6,b).
findletter_12(1,10,a).
findletter_12(1,11,b).
findletter_12(1,23,a).
findletter_12(1,28,d).
findletter_12(1,31,f).
findletter_12(1,34,h).
findletter_12(1,35,d).
findletter_12(2,6,c).
findletter_12(2,10,b).
findletter_12(2,11,c).
findletter_12(2,23,a).
findletter_12(2,28,d).
findletter_12(2,31,f).
findletter_12(2,34,h).
findletter_12(2,35,e).
findletter_12(3,6,d).
findletter_12(3,10,c).
findletter_12(3,11,d).
findletter_12(3,23,b).
findletter_12(3,28,e).
findletter_12(3,31,g).
findletter_12(3,34,i).
findletter_12(3,35,f).
findletter_12(4,6,e).
findletter_12(4,10,d).
findletter_12(4,11,e).
findletter_12(4,23,c).
findletter_12(4,28,f).
findletter_12(4,31,h).
findletter_12(4,34,j).
findletter_12(4,35,g).
findletter_12(5,6,f).
findletter_12(5,10,e).
findletter_12(5,11,f).
findletter_12(5,23,d).
findletter_12(5,28,g).
findletter_12(5,31,i).
findletter_12(5,34,k).
findletter_12(5,35,h).
findletter_12(6,6,g).
findletter_12(6,10,f).
findletter_12(6,11,g).
findletter_12(6,23,e).
findletter_12(6,28,h).
findletter_12(6,31,j).
findletter_12(6,34,l).
findletter_12(6,35,i).
findletter_12(7,6,h).
findletter_12(7,10,g).
findletter_12(7,11,h).
findletter_12(7,23,b).
findletter_12(7,28,e).
findletter_12(7,31,g).
findletter_12(7,34,i).
findletter_12(7,35,j).
findletter_12(8,6,i).
findletter_12(8,10,h).
findletter_12(8,11,i).
findletter_12(8,23,c).
findletter_12(8,28,f).
findletter_12(8,31,h).
findletter_12(8,34,j).
findletter_12(8,35,k).
findletter_12(9,6,j).
findletter_12(9,10,i).
findletter_12(9,11,j).
```

```
findletter_12(9,23,d).
findletter_12(9,28,g).
findletter_12(9,31,i).
findletter_12(9,34,k).
findletter_12(9,35,l).
findletter_12(10,6,g).
findletter_12(10,10,j).
findletter_12(10,11,k).
findletter_12(10,23,e).
findletter_12(10,28,h).
findletter_12(10,31,j).
findletter_12(10,34,l).
findletter_12(10,35,m).
findletter_12(11,6,h).
findletter_12(11,10,k).
findletter_12(11,11,l).
findletter_12(11,23,f).
findletter_12(11,28,i).
findletter_12(11,31,k).
findletter_12(11,34,m).
findletter_12(11,35,n).
findletter_12(12,6,i).
findletter_12(12,10,l).
findletter_12(12,11,m).
findletter_12(12,23,g).
findletter_12(12,28,j).
findletter_12(12,31,l).
findletter_12(12,34,n).
findletter_12(12,35,o).
findletter_2(1,15,d).
findletter_2(1,25,g).
findletter_2(1,26,c).
findletter_2(2,15,e).
findletter_2(2,25,h).
findletter_2(2,26,c).
findletter_3(1,4,a).
findletter_3(1,8,k).
findletter_3(1,21,a).
findletter_3(1,30,i).
findletter_3(1,37,n).
findletter_3(2,4,b).
findletter_3(2,8,h).
findletter_3(2,21,b).
findletter_3(2,30,j).
findletter_3(2,37,o).
findletter_3(3,4,c).
findletter_3(3,8,i).
findletter_3(3,21,c).
findletter_3(3,30,k).
findletter_3(3,37,p).
findletter_4(1,13,i).
findletter_4(1,18,m).
findletter_4(1,20,n).
findletter_4(1,32,g).
findletter_4(1,38,e).
findletter_4(2,13,j).
findletter_4(2,18,n).
findletter_4(2,20,o).
findletter_4(2,32,g).
findletter_4(2,38,g).
findletter_4(3,13,k).
findletter_4(3,18,o).
findletter_4(3,20,p).
findletter_4(3,32,h).
findletter_4(3,38,h).
findletter_4(4,13,l).
findletter_4(4,18,p).
```

```
findletter_4(4,20,q).
findletter_4(4,32,i).
findletter_4(4,38,i).
findletter_5(1,12,c).
findletter_5(1,33,k).
findletter_5(2,12,d).
findletter_5(2,33,h).
findletter_5(3,12,e).
findletter_5(3,33,i).
findletter_5(4,12,f).
findletter_5(4,33,j).
findletter_5(5,12,g).
findletter_5(5,33,k).
findletter_6(1,22,a).
findletter_6(2,22,b).
findletter_6(3,22,c).
findletter_6(4,22,d).
findletter_6(5,22,e).
findletter_6(6,22,f).
findletter_7(1,7,c).
findletter_7(1,17,e).
findletter_7(1,19,f).
findletter_7(1,36,f).
findletter_7(1,39,k).
findletter_7(2,7,d).
findletter_7(2,17,f).
findletter_7(2,19,g).
findletter_7(2,36,g).
findletter_7(2,39,l).
findletter_7(3,7,e).
findletter_7(3,17,g).
findletter_7(3,19,h).
findletter_7(3,36,h).
findletter_7(3,39,m).
findletter_7(4,7,f).
findletter_7(4,17,h).
findletter_7(4,19,i).
findletter_7(4,36,i).
findletter_7(4,39,n).
findletter_7(5,7,g).
findletter_7(5,17,i).
findletter_7(5,19,j).
findletter_7(5,36,j).
findletter_7(5,39,o).
findletter_7(6,7,h).
findletter_7(6,17,j).
findletter_7(6,19,k).
findletter_7(6,36,k).
findletter_7(6,39,p).
findletter_7(7,7,i).
findletter_7(7,17,k).
findletter_7(7,19,l).
findletter_7(7,36,l).
findletter_7(7,39,q).
findletter_8(1,5,e).
findletter_8(1,29,e).
findletter_8(2,5,f).
findletter_8(2,29,e).
findletter_8(3,5,g).
findletter_8(3,29,f).
findletter_8(4,5,h).
findletter_8(4,29,g).
findletter_8(5,5,i).
findletter_8(5,29,h).
findletter_8(6,5,f).
findletter_8(6,29,i).
findletter_8(7,5,g).
```

```
findletter_8(7,29,f).
findletter_8(8,5,h).
findletter_8(8,29,g).
findletter_9(1,1,a).
findletter_9(1,16,g).
findletter_9(1,24,b).
findletter_9(1,27,e).
findletter_9(2,1,b).
findletter_9(2,16,h).
findletter_9(2,24,b).
findletter_9(2,27,f).
findletter_9(3,1,c).
findletter_9(3,16,i).
findletter_9(3,24,c).
findletter_9(3,27,g).
findletter_9(4,1,d).
findletter_9(4,16,j).
findletter_9(4,24,d).
findletter_9(4,27,d).
findletter_9(5,1,e).
findletter_9(5,16,k).
findletter_9(5,24,e).
findletter_9(5,27,e).
findletter_9(6,1,f).
findletter_9(6,16,l).
findletter_9(6,24,f).
findletter_9(6,27,f).
findletter_9(7,1,g).
findletter_9(7,16,m).
findletter_9(7,24,c).
findletter_9(7,27,g).
findletter_9(8,1,h).
findletter_9(8,16,n).
findletter_9(8,24,d).
findletter_9(8,27,h).
findletter_9(9,1,d).
findletter_9(9,16,o).
findletter_9(9,24,e).
findletter_9(9,27,i).

difflist([]).
difflist([H|T]) :-
        diff(H,T),
        difflist(T).

diff(_,[]).
diff(H,[H1|L]) :-
        H \== H1,
        diff(H,L).




/*************************/
/* cross6               */
/* query: ?- main1.      */
/*************************/

main1 :- board(X),fail.
main1.

board([H1,H2,H3,H4,H5,H6,H7,H8,H9,H10]) :-
        sameletter88(1,H1,1,V1),
        sameletter88(1,H2,2,V1),
        sameletter68(1,H3,3,V1),
        sameletter88(1,H5,4,V1),
        sameletter28(1,H6,5,V1),
```

```
        sameletter88(1,H8,6,V1),
        sameletter88(1,H9,7,V1),
        sameletter88(1,H10,8,V1),
        sameletter88(2,H1,1,V2),
        sameletter88(2,H2,2,V2),
        sameletter68(2,H3,3,V2),
        sameletter88(2,H5,4,V2),
        sameletter28(2,H6,5,V2),
        sameletter88(2,H8,6,V2),
        sameletter88(2,H9,7,V2),
        sameletter88(2,H10,8,V2),
        sameletter84(3,H1,1,V3),
        sameletter84(3,H2,2,V3),
        sameletter64(3,H3,3,V3),
        sameletter84(3,H5,4,V3),
        sameletter83(3,H8,1,V4),
        sameletter83(3,H9,2,V4),
        sameletter83(3,H10,3,V4),
        sameletter88(4,H1,1,V5),
        sameletter88(4,H2,2,V5),
        sameletter68(4,H3,3,V5),
        sameletter88(4,H5,4,V5),
        sameletter58(1,H7,5,V5),
        sameletter88(4,H8,6,V5),
        sameletter88(4,H9,7,V5),
        sameletter88(4,H10,8,V5),
        sameletter88(5,H1,1,V6),
        sameletter88(5,H2,2,V6),
        sameletter68(5,H3,3,V6),
        sameletter88(5,H5,4,V6),
        sameletter58(2,H7,5,V6),
        sameletter88(5,H8,6,V6),
        sameletter88(5,H9,7,V6),
        sameletter88(5,H10,8,V6),
        sameletter88(6,H1,1,V7),
        sameletter88(6,H2,2,V7),
        sameletter68(6,H3,3,V7),
        sameletter88(6,H5,4,V7),
        sameletter58(3,H7,5,V7),
        sameletter88(6,H8,6,V7),
        sameletter88(6,H9,7,V7),
        sameletter88(6,H10,8,V7),
        sameletter82(7,H1,1,V8),
        sameletter82(7,H2,2,V8),
        sameletter85(7,H5,1,V9),
        sameletter55(4,H7,2,V9),
        sameletter85(7,H8,3,V9),
        sameletter85(7,H9,4,V9),
        sameletter85(7,H10,5,V9),
        sameletter88(8,H1,1,V10),
        sameletter88(8,H2,2,V10),
        sameletter18(1,H4,3,V10),
        sameletter88(8,H5,4,V10),
        sameletter58(5,H7,5,V10),
        sameletter88(8,H8,6,V10),
        sameletter88(8,H9,7,V10),
        sameletter88(8,H10,8,V10),
        difflist([H1,H2,H3,H4,H5,H6,H7,H8,H9,H10,V1,V2,V3,V4,V5,V6,V7,V8,
                V9,V10]).

sameletter18(I,J,K,L):-findletter1(I,J,M),findletter8(K,L,M).
sameletter28(I,J,K,L):-findletter2(I,J,M),findletter8(K,L,M).
sameletter55(I,J,K,L):-findletter5(I,J,M),findletter5(K,L,M).
sameletter58(I,J,K,L):-findletter5(I,J,M),findletter8(K,L,M).
sameletter64(I,J,K,L):-findletter6(I,J,M),findletter4(K,L,M).
sameletter68(I,J,K,L):-findletter6(I,J,M),findletter8(K,L,M).
sameletter82(I,J,K,L):-findletter8(I,J,M),findletter2(K,L,M).
```

```
sameletter83(I,J,K,L):-findletter8(I,J,M),findletter3(K,L,M).
sameletter84(I,J,K,L):-findletter8(I,J,M),findletter4(K,L,M).
sameletter85(I,J,K,L):-findletter8(I,J,M),findletter5(K,L,M).
sameletter88(I,J,K,L):-findletter8(I,J,M),findletter8(K,L,M).
findletter1(1,1,j).
findletter2(1,2,g).
findletter2(1,3,e).
findletter2(2,2,h).
findletter2(2,3,f).
findletter3(1,4,h).
findletter3(2,4,i).
findletter3(3,4,l).
findletter4(1,5,c).
findletter4(2,5,d).
findletter4(3,5,e).
findletter4(4,5,f).
findletter5(1,6,h).
findletter5(1,7,j).
findletter5(2,6,i).
findletter5(2,7,k).
findletter5(3,6,j).
findletter5(3,7,l).
findletter5(4,6,k).
findletter5(4,7,m).
findletter5(5,6,l).
findletter5(5,7,p).
findletter6(1,8,c).
findletter6(2,8,d).
findletter6(3,8,e).
findletter6(4,8,f).
findletter6(5,8,g).
findletter6(6,8,h).
findletter8(1,9,a).
findletter8(1,10,a).
findletter8(1,11,b).
findletter8(1,12,b).
findletter8(1,13,d).
findletter8(1,14,d).
findletter8(1,15,e).
findletter8(1,16,f).
findletter8(1,17,f).
findletter8(1,18,g).
findletter8(1,19,j).
findletter8(1,20,h).
findletter8(2,9,b).
findletter8(2,10,b).
findletter8(2,11,c).
findletter8(2,12,c).
findletter8(2,13,e).
findletter8(2,14,e).
findletter8(2,15,f).
findletter8(2,16,g).
findletter8(2,17,g).
findletter8(2,18,h).
findletter8(2,19,k).
findletter8(2,20,i).
findletter8(3,9,c).
findletter8(3,10,c).
findletter8(3,11,d).
findletter8(3,12,d).
findletter8(3,13,f).
findletter8(3,14,f).
findletter8(3,15,g).
findletter8(3,16,h).
findletter8(3,17,h).
findletter8(3,18,i).
findletter8(3,19,l).
```

```
findletter8(3,20,j).
findletter8(4,9,d).
findletter8(4,10,d).
findletter8(4,11,e).
findletter8(4,12,e).
findletter8(4,13,g).
findletter8(4,14,g).
findletter8(4,15,h).
findletter8(4,16,i).
findletter8(4,17,i).
findletter8(4,18,j).
findletter8(4,19,m).
findletter8(4,20,k).
findletter8(5,9,e).
findletter8(5,10,e).
findletter8(5,11,f).
findletter8(5,12,f).
findletter8(5,13,h).
findletter8(5,14,h).
findletter8(5,15,i).
findletter8(5,16,j).
findletter8(5,17,j).
findletter8(5,18,k).
findletter8(5,19,n).
findletter8(5,20,l).
findletter8(6,9,f).
findletter8(6,10,f).
findletter8(6,11,g).
findletter8(6,12,g).
findletter8(6,13,i).
findletter8(6,14,i).
findletter8(6,15,j).
findletter8(6,16,k).
findletter8(6,17,k).
findletter8(6,18,l).
findletter8(6,19,o).
findletter8(6,20,m).
findletter8(7,9,g).
findletter8(7,10,g).
findletter8(7,11,h).
findletter8(7,12,h).
findletter8(7,13,j).
findletter8(7,14,j).
findletter8(7,15,k).
findletter8(7,16,l).
findletter8(7,17,l).
findletter8(7,18,m).
findletter8(7,19,p).
findletter8(7,20,n).
findletter8(8,9,h).
findletter8(8,10,j).
findletter8(8,11,i).
findletter8(8,12,k).
findletter8(8,13,k).
findletter8(8,14,m).
findletter8(8,15,n).
findletter8(8,16,m).
findletter8(8,17,o).
findletter8(8,18,n).
findletter8(8,19,q).
findletter8(8,20,q).

difflist([]).
difflist([H|T]) :-
        diff(H,T),
        difflist(T).
```

```
diff(_,[]).
diff(H,[H1|L]) :-
        H \== H1,
        diff(H,L).
```

Bibliography

[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] Khayri A. M. Ali and Roland Karlsson. Scheduling speculative work in muse and perfomance results. Internal Report, SICS, 1993.

[3] Reem Bahgat. Solving Resource Allocation Problems in Pandora. Technical report, Imperial College, Department of Computing, 1990.

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

[5] Anthony Beaumont. Fuel Consumption in Andorra-I. BT review meeting, 1993.

[6] Anthony Beaumont, S. Muthu Raman, and Péter Szeredi. Scheduling Or-Parallelism in Aurora with the Bristol Scheduler. Technical Report TR-90-04, University of Bristol, Computer Science Department, March 1990.

[7] 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.

[8] 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.

[9] Johan Bevemyr, Thomas Lindgren, and Håkan Millroth. Reform Prolog: The Language and its Implementation. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 283–298. MIT Press, June 1993.

[10] Prasenjit Biswas, Shyh-Chang Su, and David Y. Y. Yun. A Scalable Abstract Machine Model to Support Limited-OR (LOR)/Restricted-AND Parallelism (RAP). In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1161–1180. MIT Press, June 1988.

[11] Peter Borgwardt. Parallel Prolog using Stack Segments on Shared-Memory Multiprocessors. In *Proceedings of the 1984 International Logic Programming Symposium*, pages 2–11, February 1984.

[12] Koen De Bosschere and Jean-Marie Jacquet. Multi-Prolog: Definition, Operational Semantics and Implementation. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 299–313. MIT Press, June 1993.

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

[14] J. Briat, M. Favre, C. Geyer, and J. Chassin. Scheduling of or-parallel Prolog on a scaleable, reconfigurable, distributed-memory multiprocessor. In *Proceedings of Parallel Architecture and Languages Europe*. Springer Verlag, 1991.

[15] 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.

[16] 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.

[17] Mats Carlsson and Johan Widen. SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science, 1988. SICS Research Report R88007B.

[18] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[19] Takashi Chikayama. Operating system PIMOS and kernel language KL1. In ICOT [75], pages 73–88.

[20] Andrzej Ciepielewski and Seif Haridi. A Formal Model for Or-Parallel Execution of Logic Programs. In *IFIP 83 Conference*, pages 299–305. North Holland, 1983.

[21] Andrzej Ciepielewsky, Khayri A. M. Ali, and Prasenjit Biswas. Execution Models and Parallel Implementations of Prolog. unpublished, June 1993.

[22] Keith L. Clark, F. G. McCabe, and S. Gregory. IC-PROLOG – language features. In K. L. Clark and S. Å. Tärnlund, editors, *Logic Programming*, pages 253–266. Academic Press, London, 1982.

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

[24] William Clocksin and Chris Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[25] Jacques Cohen. A View of the Origins and Development of Prolog. *Communications of the ACM*, 31(1):26–36, 1988.

[26] J. S. Conery. *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, University of California at Irvine, 1983.

[27] J. S. Conery. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, 1987.

[28] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I Compilation. *New Generation Computing*, to appear.

[29] Barry Crabtree. A clustering system to network control, british telecom, March 1991.

[30] J. A. Crammond. A Comparative Study of Unification Algorithms for OR-Parallel Execution of Logic Languages. In D. DeGroot, editor, *Int. Conf. on Parallel Processing*, pages 131–138, St. Charles, Ill., August 1985. IEEE.

[31] 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.

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

[33] 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.

[34] Mark E. Crovella and Thomas J. Leblanc. Performance Debugging using Parallel Performance Predicates. Technical report, Dept. of Computer Science, University of Rochester, June 1992.

[35] 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.

[36] Doug DeGroot. Restricted And-Parallelism. In Hideo Aiso, editor, *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, pages 471–478. Institute for New Generation Computing, Tokyo, 1984.

[37] Doug DeGroot. A Technique for Compiling Execution Graph Expressions for Restricted And-Parallelism in Logic Programs. In C. Polychronopoulos, editor, *Lecture Notes in Computer Science 297, Supercomputing*, pages 1074–1093, 1987.

[38] A. K. Dewdney. Mathematical Recreations – A Compendium of Math Abuse from around the World. *Scientific American*, 263(5):142–145, September 1990.

[39] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *International Conference on Fifth Generation Computer Systems 1988*, pages 693–702. ICOT, Tokyo, Japan, November 1988.

[40] Terrence Disz, Ewing Lusk, and Ross Overbeek. Experiments with OR-Parallel Logic Programs. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, pages 576–600. MIT Press, 1987.

[41] Kees Doets. *From Logic to Logic Programming*. MIT Press, 1994. Foundations of Computing Series.

[42] M. Dorochevsky, L. L. Li, M. Reeve, K. Schuerman, and A. Veron. Elipsys – A Parallel Programming System Based on Logic. In *Lecture Notes in Artificial Intelligence*, volume 624, pages 469–471. Springer-Verlag, 1992.

[43] Inês 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.

[44] 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.

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

[46] M. Falaschi, Y. Kesten, and E. Shapiro. Andorra Prolog in FCP(:). In *6o Convegno Sulla Programazione Logica, GULP91, Pisa*, pages 521–537, June 1991.

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

[48] Wai-Keong Foong. Or-Parallel Prolog with Heuristic Task Distribution. In *Lecture Notes in Artificial Intelligence 592, Logic Programming Russian Conference*, pages 193–200, 1991.

[49] 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.

[50] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, 1988.

[51] Luis Gómez, Manuel Carro, and Manuel Hermenegildo. Some Paradigms for Visualising Parallel Execution of Logic Programs. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 184–202, 1993.

[52] Antonio González and Jordi Tubella. The Multipath Parallel Execution Model for Prolog. In *Proceedings of PASCO'94 (Parallel Symbolic Computation Conference)*, pages 426–432. World Scientific Publishing, 1994.

[53] 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.

[54] Steve Gregory. *Parallel Logic Programming in PARLOG*. Addison–Wesley, 1987.

[55] Steve Gregory. Experiments with Speculative Parallelism in PARLOG. In *Proceedings of the 1993 International Logic Programming Symposium*, pages 370–387. MIT Press, 1992. Also available as technical report from the Department of Computer Science, University of Bristol, CSTR-93-05.

[56] Gopal Gupta. A Timestamp Based Technique for Dynamic Parallel Evaluation of Cross Product Sets. Research Report, Department of Computer Science, University of Bristol, April 1990.

[57] Gopal Gupta. *Multiprocessor Execution of Logic Programs*. PhD thesis, Department of Computer Science, New Mexico State University, Las Cruces, July 1993.

[58] 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.

[59] Gopal Gupta, M. V. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proceedings of the Eleventh International Conference on Logic Programming*, Italy, June 1994.

[60] Gopal Gupta and Bharat Jayaraman. On Criteria for Or-parallel Execution of Logic Programs. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 737–756. MIT Press, October 1990.

[61] Gopal Gupta, Enrico Pontelli, and Manuel Hermenegildo. &ACE: A High Performance Parallel Prolog System. In *Proceedings of the First International Symposium on Parallel Symbolic Computation, PASCO'94*, 1994.

[62] 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.

[63] 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.

[64] Gopal Gupta and David H. D. Warren. An Interpreter for the Extended Andorra Model. Technical Report, University of Bristol, 1991.

[65] Seif Haridi and Per Brand. Andorra Prolog–an integration of Prolog and committed choice languages. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*. ICOT, 1988.

[66] Bogumił Hausman. Pruning and Scheduling Speculative Work in Or-Parallel Prolog. In *PARLE 89, Conference on Parallel Architectures and Languages Europe*. Springer-Verlag, 1989.

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

[68] Manuel Hermenegildo. An Abstract Machine for Restricted And-Parallel Execution of Logic Programs. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 25–39. Springer-Verlag, 1986.

[69] 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.

[70] P. M. Hill and J. W. Lloyd. *The Göedel Programming Language*. MIT Press, Logic Programming Series, 1994.

[71] R. Hill. LUSH-Resolution and its Completeness. Dcl memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.

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

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

[74] Avshalom Houri and Ehud Shapiro. A sequential abstract machine for flat concurrent Prolog. *The Journal of Logic Programming*, 7(2):85–124, September 1989.

[75] Japan ICOT, editor. *Proceedings of the International Conference on Fifth Generation Computer Systems*. Association for Computing Machinery, 1992.

[76] Sverker Jansson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *Proceedings of the 1991 International Logic Programming Symposium*, pages 167–186. MIT Press, October 1991.

[77] B. Jayaraman and D. A. Plaisted. Programming with Equations, Subsets, and Relations. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the 1989 North American Conference on Logic Programming*, pages 1051–1068, 1989.

[78] L. V. Kalé. The REDUCE OR Process Model for Parallel Evaluation of Logic Programming. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 616–632, 1987.

[79] Robert A. Kowalski. The Early Years of Logic Programming. *Communications of the ACM*, 31(1):38–43, 1988.

[80] Takashi Kurozumi. Overview of the ten years of the FGSC project. In ICOT [75], pages 9–19.

[81] Ted G Lewis and Resham El-Rewini. *Introduction to Parallel Computing*. Prentice–Hall, 1992.

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

[83] Y-J. Lin and V. Kumar. An Execution Model for Exploiting And-Parallelism in Logic Programs. *New Generation Computing*, 5(4):393–425, 1988.

[84] Y-J. Lin and V. Kumar. And- Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: a Summary of Results. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*. MIT Press, 1988.

[85] Zheng Lin. Self-Organising Task Scheduling for Parallel Execution of Logic Programs. In *Proceedings of the 1992 International Conference on Fifth Generation Computer Systems*, pages 859–868. ICOT, 1992.

[86] G. Lindstrom. Or-Parallelism on Applicative Architectures. In *Proceedings of the 1984 International Logic Programming Symposium*, pages 159–170, July 1984.

[87] J. W. Lloyd. Combining Functional and Logic Programming Languages. In *Proceedings of the 1994 International Logic Programming Symposium*, pages 43–57. MIT Press, 1994.

[88] 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.

[89] Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[90] C McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared Memory Multiprocessors. Technical Report TR-90-03-02, University of Washington, Department of Computer Science and Engineering, March 1990.

[91] 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.

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

[93] Remco Moolenaar and Bart Demoen. Hybrid tree search in the Andorra Model. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 110–123. MIT Press, June 1994.

[94] C. Moss and A. Roth. The Prolog 1000 Database. Available through anonymous ftp from src.doc.ac.uk in packages/prolog-progs-db/prolog1000.v1 or from www, August 1993.

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

[96] K. Muthukumar and M. V. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[97] Lee Naish. Parallelizing NU-Prolog. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1546–1564. MIT Press, August 1988.

[98] G. H. Pollard. *Parallel Execution of Horn Clause Programs*. PhD thesis, Imperial College, London, 1981. Dept. of Computing.

[99] T.J. Reynolds, A. J. Beaumont, A. S. K. Cheng, S. A. Delgado-Rannauro, and L. A. Spacek. BRAVE – A Parallel Logic Language for Artificial Intelligence. *Fifth Generation Computer Systems*, 4:69–75, 1988.

[100] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(23):23–41, January 1965.

[101] 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.

[102] 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.

[103] 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.

[104] Vijay A. Saraswat, Ken Kahn, and Jacob Levy. Janus: A step towards distributed constraint programming. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 431–446, 1990.

[105] M. Sato, H. Shimizu, A. Matsumoto, K. Rokusawa, and A. Goto. KL1 Execution Model for PIM Cluster with Shared Memory. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, pages 338–355. University of Melbourne, "MIT Press", May 1987.

[106] David C. Sehr and Laxmikant V. Kalé. Estimating the Inherent Parallelism in Prolog Programs. In *Proceedings of the 1992 International Conference on Fifth Generation Computer Systems*, pages 783–790. ICOT, 1992.

[107] Jan Serr. Scheduling in ParAKL. Master's thesis, Catholic University of Leuven, Belgium, 1993.

[108] Kish Shen. Prometheus: An And/Or Parallel Prolog — A High-level View. Technical Report TR-91-39, Computer Science Department, University of Bristol, 1991.

[109] Kish Shen. Exploiting And-parallelism in Prolog: the Dynamic Dependent And-parallel Scheme (DDAS). In *Logic Programming: Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 717–731, 1992.

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

[111] Kish Shen. Memory Management and Goal Scheduling for Parallel Execution Revisited. Internal Report, Bristol University, 1993.

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

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

[114] Donald A. Smith. MultiLog: Data Or-Parallel Logic Programming. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 314–331. MIT Press, 1993.

[115] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, 1986.

[116] G. Succi, G. Marino, G. Colla, D. Co, S. Novella, A. Pata, A. Regoli, and L. Vigano. SEL Compiler and Abstract Analyzers. In *ALPUK92, City University, London*, pages 132–144, 1992.

[117] M. Sugie, M. Yoneyama, N. Ido, and T. Tarui. Load-Dispatching Strategy on Parallel Inference Machines. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 987–993. ICOT, 1988.

[118] Peter Szeredi. The Msec Interface for Aurora. Internal Report, March 1988.

[119] 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.

[120] 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.

[121] T. Chikayama, T. and Fujise, and H. Yashiro. A Portable and Reasonably Efficient Implementation of KL1. In *Proceedings of the Eleventh International Conference on Logic Programming*, June 1993.

[122] Yasutaka Takeda, Hiroshi Nakashima, Kanae Masuda, Takashi Chikayama, and Kazuo Taki. A Load Balancing Mechanism for Large Scale Multiprocessor Systems and its Implementation. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 978–986. ICOT, 1988.

[123] Jiro Tanaka, Kazunori Ueda, Toshihiko Miyazaki, Akikazu Takeuchi, Yuji Matsumoto, and Koichi Furukawa. Guarded Horn Clauses and Experiences with Parallel Programming. In *1986 Proceedings Fall Joint Computer Conference*, pages 948–954. IEEE Computer Society Press, November 1986.

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

[125] Andrew Tucker and Anoop Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. *ACM Sigplan*, pages 159–166, 1989.

[126] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 140–156. MIT Press, Cambridge MA, 1987.

[127] Kazunori Ueda and Masao Morita. A New Implementation Technique for Flat GHC. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 3–17. MIT Press, June 1990.

[128] Michel van Caneghem and David H. D. Warren, editors. *Logic Programming and its Applications*. Ablex Publishing Corporation, 1986.

[129] R. J. van der Pas and J. M. van Kats. Parallelism in a multi-user environment. *Parallel Computing*, 17:285–296, 1991.

[130] Pascal van Hentenryck. *Constraint Satisfaction in Logic programming*. MIT Press, 1989.

[131] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.

[132] P. Van Roy. 1983–1993–The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 20:385–441, 1994.

[133] P. Van Roy, B. Demoen, and Y. D. Willems. Improving the execution Speed of compiled Prolog with Modes, Clause Selection and Determinism. In *TAPSOFT'87*, pages 111–125. Springer Verlag, 1987.

[134] D. S. Warren. Efficient Prolog Memory Management for Flexible Control Strategies. *New Generation Computing*, 2:361–369, 1984.

[135] David H. D. Warren. *Applied Logic—Its Use and Implementation as a Programming Tool*. PhD thesis, Edinburgh University, 1977. Available as Technical Note 290, SRI International.

[136] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[137] David H. D. Warren. Or-Parallel Execution Models of Prolog. In *TAPSOFT'87, The 1987 International Joint Conference on Theory and Practice of Software Development, Pisa, Italy*, pages 243–259. Springer-Verlag, March 1987.

[138] 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.

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

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

[141] David H. D. Warren and Fernando C. N. Pereira. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. Technical Note, Dept of AI, University of Edinburgh, 1981.

[142] Harald Westphal, Philippe Robert, Jacques Chassin, and Jean-Claude Syre. The PEPSys model: combining backtracking, and- and or-parallelism. In *The 1987 Symposium on Logic Programming, San Francisco, California*. IEEE, 1987.

[143] R. Yang, I. Dutra, and V. Santos Costa. Design of the Andorra-I System. Internal Report, PEPMA Project, January 1991.

[144] Rong Yang. *P-Prolog a Parallel Logic Programming Language*, volume 9. World Scientific, Singapore, 1987. Series in Computer Science.

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

[146] 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.

[147] 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.

[148] 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.

[149] Gao Yaoqing, Wang Dingxing, Qiu Xiaolin, Zheng Weiming, Shen Meiming, and Hu Shouren. Intelligent Scheduling And- and Or- Parallelism in the Parallel Logic Programming System RAP/LOP-PIM. In *Proceedings of the 1991 International Conference on Parallel Processing*, August 1991.

[150] 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.

[151] 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.