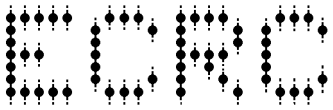


7 November 1995

The Handbook of Parallel Constraint Logic Programming Applications

Alexander Herold (ed.) (ECRC)

Abstract

This is the final technical deliverable of the APPLAUSE project. It provides a summary of the APPLAUSE technology and its implementation in ElipSys and ECL^{i}PS^e. It then includes a tutorial on how to use constraints and parallelism in ECL^{i}PS^e. The main part of the book contains descriptions of the APPLAUSE applications summarizing the experience gathered in the course of the project. The book is targeted at potential users of parallel constraint logic programming (CLP) and ECL^{i}PS^e. It aims to encourage them to exploit the potential of parallel CLP by presenting the APPLAUSE exemplars.

European Computer-Industry
Research Centre GmbH
Arabellastr. 17,
D-81925 München
Germany

Contents

Chapter 1.

The APPLAUSE Project

Alexander Herold

1 The APPLAUSE Framework

During the last decade a new programming paradigm called *"logic programming"* has emerged. The best known representative of this new class of programming languages is *Prolog*, originated from ideas of Alain Colmerauer in Marseille and Bob Kowalski in Edinburgh. Programming in Prolog differs from conventional programming both stylistically and computationally, as it uses logic to declaratively state problems and deduction to solve them. Hence logic programming belongs to the class of declarative programming languages.

It is a truism that logic programming (LP) is no panacea to solve all problems. On the contrary LP needs to be extended to provide a useful tool for solving real-life problems. In this report we will concentrate on solving combinatorial problems, such as scheduling and planning problems, resource allocation problems or problems arising in decision support, ie. this report will focus on a technology to solve large-scale search problems. To meet the requirements of such problems two essential extensions to the basic paradigm of logic programming were necessary.

First logic programming was extended by the concept of constraints. In particular the introduction of so called finite domains constraints made it possible to solve large combinatorial search problems. The essential idea was to use the constraints to prune the search space in an a priori way, thus shifting the basic search paradigm of logic programming from a "generate and test" approach to a so called "constrain and generate" approach. With this new search paradigm the main drawback of "generate and test" is avoided, ie. to repeatedly generate candidate solutions which are later rejected. Very often combinatorial problems occur in optimization tasks. In order to cope with such combinatorial optimization problems some constraint logic programming systems offer optimization strategies based on branch & bound techniques.

However, for real-life problems the remaining search space which needs to be explored can still be very large. Hence it is quite obvious to exploit the inherent parallelism of such a search procedure. In such a way constraints and parallelism complement each other perfectly. Constraints are pruning the search space a priori and parallelism is speeding up the search of what remains. Combined with branch & bound techniques even super-linear speedups can be observed. In terms of logic programming parallelism supporting search is called OR-parallelism. The introduction of OR-parallelism into the constraint logic programming framework is resulting in a parallel constraint logic system.

These were horizons with which the first parallel CLP system was conceived. It was called ElipSys. The initial development of ElipSys was carried out within the ESPRIT project EDS (European Declarative System) between 1989 and 1993. The goal of the EDS project was to develop a large-scale parallel database server equipped with two declarative programming languages, one based on LISP and one based on Prolog. ECRC was responsible for the Prolog based system and developed the parallel CLP system ElipSys. ElipSys was designed with the aim of providing a high level tool for the exploitation of the processing power being delivered by the new generation of general purpose multi-processors, with a particular focus on real-world and real-size problems in Operation Research and Artificial Intelligence. In this respect, it was conceived as a valid alternative to imperative languages and their parallel extensions, since it relieves the programmer of the low-level

tactical concerns associated with parallel programming and allows him/her to concentrate on the high-level strategic issues (eg. algorithm design). At the end of the EDS project ElipSys was at a first stage of maturity as a practical parallel programming system and it was available on various workstations and multiprocessors.

Initial feedback from users within the EDS project highlighted the need for further development of the ElipSys environment. Since ElipSys was aimed at being a practical system collaboration and understanding of users' needs was crucial for its future effectiveness. These were the motivations which lead to launch the ESPRIT project APPLAUSE (Application & Assessment of Parallel Programming Using Logic). The initial platform on which APPLAUSE was built was ElipSys. In the course of the project ECRC decided to integrate its different logic programming systems into one leading edge system, called ECL^{i}PS^e. In particular, ECL^{i}PS^e integrates the ElipSys technology and became the new supporting platform for the APPLAUSE project.

The main objective of the APPLAUSE project was to support the emergence of parallel CLP as a leading programming technology and its implementations ElipSys / ECL^{i}PS^e as the corresponding programming systems by combining the efforts of the providers of this important European technology with those of a set of end-users and application developers well positioned in commercially important and challenging application areas. The approach adopted in the APPLAUSE project, to achieve the objective of moving parallel CLP to the market, was to build a number of credible demonstrations of its use.

For this purpose, the APPLAUSE project has selected three commercially important generic classes of applications and within each of these classes it has developed exploitable exemplars:

- Planning & Scheduling
- Decision Support
- Multi-Agent Systems

For Planning and Scheduling the project has built two major demonstrators for the Aircraft and Space Industry. The first one enhancing an existing planning system for aircraft production originally implemented in the sequential CLP system CHIP. The second application is a planning system for constructing optimized curricula for the training of aircraft pilots.

For Decision Support, the project addressed two different application domains: Molecular Biology and Environmental Monitoring and Control. In the former, two application domains have been considered. First, a system for protein sequencing and structure analysis has been developed and second a system for Genetic Map Construction. In the area of Environmental Monitoring and Control, a pilot application for the evaluation, simulation and control of the pollution in the Venice Lagoon was developed.

For Multi-Agent Systems, the project focused on the Tourism industry. A tourist advisor for Greece was conceived.

This report is structured into seven chapters. In this first chapter we will review the main technological concepts and the underlying platforms of the APPLAUSE project.

The purpose of this chapter is provide enough insight to understand the technological basis of this report. The second chapter contains a tutorial on parallelism and constraints in ECL^{i}PS^e. This is a tutorial for the working programmer on the ECL^{i}PS^e parallel constraint logic programming language. It assumes previous experience of ECL^{i}PS^e, or at least some version of Prolog, and introduces the parallelism and constraints features. Further details can be found in the ECL^{i}PS^e User Manual [ECL95] and the ECL^{i}PS^e Extension User Manual [ECL94]. The remaining chapters contain descriptions of the following APPLAUSE applications:

- PSAP: A Planning System for Aircraft Production;
- TCO: Training Curriculum Optimization;
- Using Parallel CLP to Predict Protein Topology;
- A Decision Support System for the Venice Lagoon;
- MaTourA: Multi-Agent Tourist Advisor.

2 The APPLAUSE Technology

The power of logic programming languages rests on three mechanisms: **unification, relational form and nondeterministic computation.** Constraint logic programming is contributing to all these aspects: it extends unification to constraint solving in **new constraint domains** richer than the usual "Herbrand Universe". It allows terms **more expressive** than uninterpreted Herbrand terms to be handled, and it provides **new computation rules** overcoming the limitations of chronological backtracking as used in logic programming.

The CLP scheme has several advantages over traditional logic programming languages. As far as programming convenience is concerned, it allows the programmer to reason directly in terms of the intended domain of discourse instead of forcing the coding of semantic objects in terms of a Herbrand universe. As far as efficiency is concerned, it allows implementors to exploit the properties of the new computation domains in order to devise efficient constraint-solving algorithms. Hence CLP combines the declarativeness and flexibility of logic programming with the efficiency of conventional approaches.

One of the first CLP systems was CHIP [DVS+88b] developed at ECRC between 1986 and 1990. The CLP roots of the APPLAUSE technology are going back to this system. CHIP was designed to tackle real world constrained search problems. It was based on the concept of active use of constraints [Gal85], [Din86], [Van89a] and included three new computation domains: finite domain restricted terms, boolean terms and linear rational terms. For each of them CHIP used specialized constraint solving techniques: consistency techniques for finite domains, equation solving in the boolean algebra and a symbolic simplex-like algorithm for linear constraints. CHIP was already successfully applied to a large number of industrial applications [DVS+88a]. Below the computation domains of CHIP relevant to the APPLAUSE applications are introduced:

Finite Domains: The basic feature of CHIP for solving discrete combinatorial problems is the ability to work on domain-variables, i.e. variables ranging over a finite set of values (e.g., finite set of natural numbers). CHIP provides a large variety of constraints on domain-variables:

- It can cope with arithmetic constraints such as equations, disequations and inequalities over arithmetic terms constructed from natural numbers, domain-variables over the natural numbers and the usual arithmetic operators.
- It allows symbolic constraints on domain-variables to express logical or functional relationships.
- It also includes some extensions for finding solutions optimizing some evaluation functions based on branch and bound techniques. These meta-predicates can be used for solving combinatorial optimization problems.

All constraints are solved through consistency checking and constraint propagation techniques, a powerful paradigm emerging from AI to solve discrete combinatorial problems. The computational framework for integrating consistency techniques into logic programming has been defined in [Van89a].

Rational Arithmetic: Rational terms are built from rational numbers, rational variables (i.e., variable ranging over rational numbers) and the following arithmetic operators: addition, subtraction and multiplication by a constant (therefore restricted to linear terms). Constraints allowed on rational terms are equations, inequalities, and disequations. The constraint-solver is an adaptation of the simplex algorithm based on variable elimination, and not on matrix manipulations. CHIP can be used for deciding if a set of constraints is satisfiable or not, and for finding the most general solution to a set of constraints optimizing a linear evaluation function [DVS+88b].

Other computation domains include boolean constraints [BS87], nonlinear arithmetic constraints [Hon92] and sets constraints [Ger94].

Several of the above mentioned techniques make use of a data-driven computation. Moreover CHIP offers a number of constructs to explicitly control the execution in a demon-driven way. Most importantly CHIP contained a delay mechanism which enables coroutining in a demon-driven way.

One of the main limitations of such "traditional" constraint logic programming systems is the limitation, that the users are bound to use the constraints offered by the systems. During the last years an important research effort has been spent on developing new concepts to support the user in writing his own constraints. Essentially three different approaches following different philosophies have been proposed:

- low-level predicates and explicit programming of constraint propagation [MS92]
- constraint handling rules [Fru95]
- generalized propagation and approximate generalized propagation [LW93]

The field of CLP has been extensively reviewed during the last years. An informal introduction to the different concepts mentioned above can be found in [FHK+92]. The reader interested in a thorough technical review is referred to [JM94]. The concept of finite domains which is the most important one for the APPLAUSE applications is also introduced in the next chapter.

One of the main advantages of LP and CLP is that there is a great deal of implicit parallelism expressed in most programs. Essentially there are two sources of parallelism in a logic program. Firstly two goals can be executed in parallel. In this case we speak of AND-parallelism. In case the two goals are independent, ie. they do not share any information, in other words they do not have any variables in common, we are exploiting independent AND-parallelism otherwise it is dependent AND-parallelism. Secondly two different alternatives of one goal can be explored concurrently. In this case we are exploiting OR-parallelism. In particular, OR-parallelism is most beneficial to search-based applications. It was therefore natural to introduce OR-parallelism into the CLP framework [Van89b].

Ideally, the programmer should not have to think about the parallelism in his program at all, but only think of what needs to be done *sequentially*. However, we are still quite far from this ideal, though there has been considerable research into automatic parallelization. In the current state of the art not all parallelization decisions can be made by the system, and the best parallelizations are still found by humans. This was the main reason for providing the user with the possibility to annotate the program, thus to giving the system directives of which parts of the search tree to explore in parallel. A programmer typically uses knowledge about the program to make an initial parallelization, and then measures speedups associated with different parallelizations on a set of queries. But finding a good parallelization of a program can be a difficult task. The tutorial in Chapter 2 gives hints on how to tackle this task and the applications descriptions summarizes the experiences made throughout the course of the APPLAUSE project.

In addition to annotating his program the programmer has a set of primitives at his disposal which are providing efficient parallel versions of built-ins typically used in search based programs. In combinatorial search problems very often a cost is associated to each solution and the user is usually interested in finding a solution with an optimal cost, that is one with the lowest possible cost. Most CLP systems provide functions for optimization based on the branch and bound method as also mentioned above. In a nutshell, branch and bound works by searching for a solution to the problem and then adding a further constraint that any new solution must be better than the current best. This strategy fits well with the standard backtracking search employed by most sequential logic programming systems.

However, experiments have shown (partly carried out in the framework of APPLAUSE) that the current approaches to parallelizing branch and bound in CLP are still not able to solve many of the larger optimization problems in reasonable time spans (such as some very complex job-shop scheduling problems). More recently a new application of parallelism has been developed which is called cost-parallelism [PM95]. Cost-parallelism has been combined with Or-parallelism to find optimal solutions more quickly than pure Or-parallel branch and bound, and also find more accurate solutions within a given time frame. First experiments with cost-based optimization in some of the APPLAUSE applications have shown promising results.

OR-parallelism and constraints are complementary means to attack search based problems. Because of the NP-completeness of the target problems, solutions cannot avoid search. They should mainly rely on the effective modeling of the problem with the appropriate data structures and constraints, so as to have as low a complexity as possible. Search is used to explore the remaining cases which have not been excluded. OR-parallelism is then a natural way to speed-up this phase of the computation. In other words, the computations after the "don't know point" inherent to the solving of NP-complete problems can be naturally supported by "don't know" parallelism, that is OR-parallelism.

3 The APPLAUSE Systems

In the following we will briefly introduce the initial APPLAUSE platform ElipSys and its successor, the ECL^{i}PS^e system.

3.1 ElipSys

The general objective of the ElipSys [VSRL93] project was the development of a technology which enables the development and execution of *portable high performance applications* dealing with *very large search spaces*. Being convinced that there is no single panacea for parallel processing, ElipSys is targeted at a specific class of applications (large search problems) running on general purpose parallel machines.

The ElipSys language has been designed to effectively support the development and execution of real-world search applications. The ElipSys language uses a logic-based notation to describe search spaces. It uses constraints and parallel logic programming as two complementary means to control the exploration of search spaces. Constraints are used to *a priori* reduce the search space. Parallelism is used to explore the different alternatives in parallel when a "don't know" point in the search space is encountered.

From its CHIP [DVS+88b] ancestor ElipSys has inherited finite domains, built-in arithmetic and symbolic constraints. It has been equipped with a flexible user level language integrating in one environment finite domains, built-in arithmetic constraints, advanced coroutining mechanisms and term manipulation primitives. While the first two items allow a CHIP programming style and expressive power, the last two make it possible to let users develop at user-level additional constraints or add new-inference rules.

The ElipSys language is equipped with an annotation which enables the programmer to express where the search tree described by the program can be traversed in parallel. The annotation is a conservative hint to the underlying parallel system which is free to exploit it or not. No splitting can occur if no annotation is present. In addition ElipSys provides specialized primitives for the parallel enumeration of finite domains.

ElipSys is equipped with side-effects and cut to support the sequential Prolog-style code which typically surrounds a pure constraint logic programming piece of search code (which can use parallel execution) in real-world applications. During the parallel execution of the search-based core of the application, cuts behave like cavalier cuts. Side-effects are fully asynchronous. In the same spirit, collecting predicates such as *bagof/3* and the like do not

behave as in a sequential system and gather solutions in the order in which the system finds them.

Many combinatorial problems involve the optimization of one criteria. Optimization constructs are higher-order predicates which receive a goal and a cost function as arguments. ElipSys offers optimization predicates well-integrated into the parallel constraint environment.

Execution models for OR-parallel logic programming have been studied extensively. The various proposals differ in binding environment, scheduling, and pruning. Most execution models are targeted at either a shared store machine or a distributed store machine. The ElipSys execution model should however enable efficient implementations on both kinds of parallel machines.

There are basically three different options for the binding environment: *sharing*, *copying*, and *recomputation*. ElipSys uses a shared binding environment based on binding arrays. Scheduling and pruning involve synchronization and fine grain data transfer. ElipSys has therefore a message based scheduler. The dictionary and code are seldomly updated and are easy and efficient to implement with shared memory. On distributed store machines the ElipSys system relied on the existence of virtual shared memory.

3.2 ECL^{i}PS^e

During the course of the APPLAUSE project ECRC launched its ECL^{i}PS^e project. It aims brings together the experience in designing and developing logic programming system that ECRC has gained during its 10 years of existence. A wide range of systems closely or loosely related to the logic programming paradigm have been studied, implemented and experimented with. To mention a few highlights:

- Efficient implementation of LP systems; incremental compilers; stable and robust product-quality systems; development of sophisticated programming environments, e.g. debugging support and graphical interfaces
- Extensions of logic programming, in particular constraint programming and object-oriented programming; rapid prototyping and efficient implementation of such extensions; both application-driven and research-based extensions were experimented with.
- Deductive databases, complementing relational database technology with deductive capabilities; manipulation, storage and retrieval of complex structures in a database context.
- Exploitation of parallel platforms for efficient execution of constraint search problems.

The core of the ECL^{i}PS^e system is based on SEPIA, an efficient product-quality Prolog system which was designed as an efficient support for various LP extensions. It comprises a large set of features that allow new extensions and systems to be efficiently implemented and interfaced.

The database component of ECL^{i}PS^e has been taken from the MegaLog system. MegaLog's database subsystem has been directly interfaced to ECL^{i}PS^e. Currently ECL^{i}PS^e is used as the technological basis for ECRC's contribution to the ESPRIT project IDEA.

The constraint component of ECL^{i}PS^e is based on the CHIP technology. However, the architecture of the constraint component has been completely redesigned to support the easy integration of new constraint solvers and constraint systems.

Finally ECL^{i}PS^e integrates the parallel technology of the ElipSys system. Thus the ElipSys users, in particular those in APPLAUSE, benefited from the rich constraint libraries offered in ECL^{i}PS^e. It should be mentioned that using ElipSys the TCO application would not have been possible.

The Constraint System of ECL^{i}PS^e

Two major requirements influenced the Constraint System of ECL^{i}PS^e. First it had to be backward compatible with the original CHIP and ElipSys systems to support the application development which was carried out in the framework of the ESPRIT projects CHIC and APPLAUSE. Second it had to support and integrate the novel research prototypes developed at ECRC within the CHIC project. It became quickly clear that it was beyond the capabilities of the existing systems to satisfy such divers needs in terms of functionality, extensibility and flexibility.

The main contribution of ECL^{i}PS^e in the area of CLP architecture is the design and implementation of its novel generic constraint interface capable of supporting the requirements coming from both applications and research. This constraint interface rests on two fundamental concepts:

- metaterms and
- suspensions

Metaterms are a generic means to program rich data structures, eg. those required to implement new computation domains of constraint logic programming languages. The second concept, ie. suspensions, is a generalization of a well-known mechanism in logic programming to overcome the left-to-right depth-first computation rule of Prolog-like languages. Its main purpose is to provide a high-level interface to program constrained search techniques.

The constraint interface is now fully integrated into the ECL^{i}PS^e system providing a uniform high-level interface for the development of the different constraint extensions. These constraint extensions are implemented in the ECL^{i}PS^e language itself as libraries. The interoperability of these libraries, a major problem for other constraint systems, is automatically guaranteed. ECL^{i}PS^e now provides the following constraint libraries:

- finite domains
- linear rational arithmetic

- set constraints
- generalized propagation
- constraint handling rules

The sequential architecture of the ECL^{i}PS^{e}system is sketched in figure 3.1.

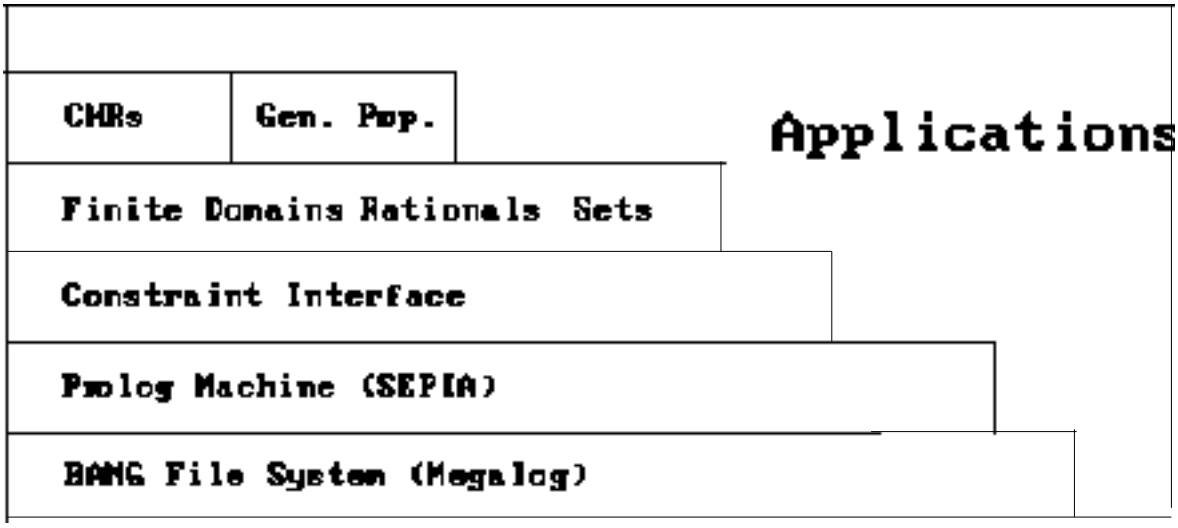


Figure 3.1: Sequential architecture of ECL^{i}PS^e

Parallel ECL^{i}PS^e

Parallel ECL^{i}PS^e is an extension of the current sequential ECL^{i}PS^e system exploiting OR-parallelism. It replaces ElipSys and hence had to support existing ElipSys applications. The main criteria used in the design were the following:

1. **Execution Platforms:** The system should execute efficiently on a wide variety of platforms, from true shared-memory multiprocessors to a network of heterogeneous workstations. The main implication of this criteria is that the design cannot make any assumptions about the availability of shared memory and even virtual shared memory.
2. **ElipSys Integration:** The system should incorporate ElipSys features, such as optimized constraints handling and visualization tools which have proven to be very useful to end-users.
3. **Modular Design :** The sequential engine should be modified as little as possible so as to retain its efficiency and functionality in a parallel setting.

Given the above requirements it became quickly clear that parallel ECL^{i}PS^e had to deviate from the binding array scheme of ElipSys, since this would have required major changes to the existing sequential system. In addition, efficient implementation of the binding

arrays scheme will require support for virtual shared memory which is not available on all parallel platforms.

A realistic scenoario of a common computing environment in the near future is a network of workstations where some or all workstations are multi-processors with a small number ($2-8$) of CPUs. Based on the results of a first protoype for parallel CLP [MS94] on such a network of heterogeneous workstations, it was decided that ECL^{i}PS^e will use a hybrid scheme:

- It is using stack-copying when sharing work between processes on the same machine, ie. for shared multi-processor machines Parallel ECL^{i}PS^e is built on top of copying scheme.
- However, when sharing work across machines Parallel ECL^{i}PS^e is using a recomputation model, as it turned out that it is cheaper to recompute than to communicate.

The idea of such a hybrid copying/recomputation scheme is illustrated in figure 3.2. The dashed line around the engine processes E on the multiprocessor indicates that they share some state. (such as global dictionaries).

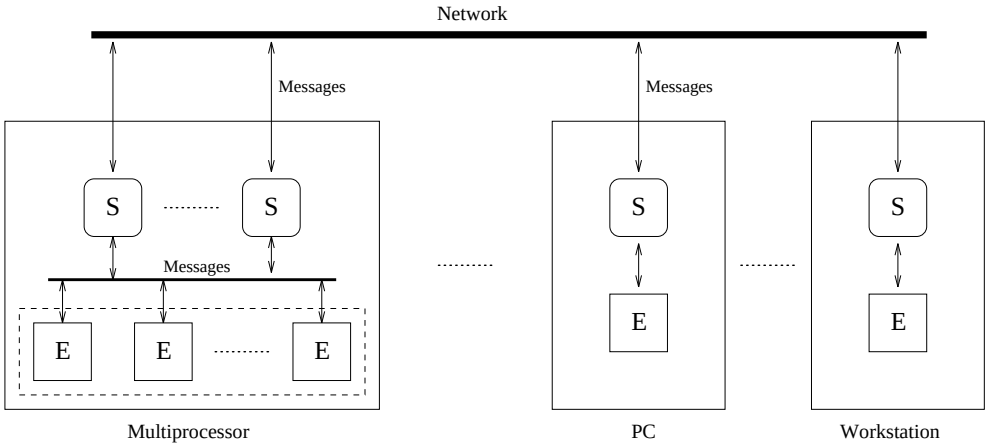


Figure 3.2: Hybrid Model

The main issue in such a system is scheduler, which has to coordinate the concurrent activities of the engines so that parallel CLP applications are run correct and fast. It also has to decide on the fly whether stack-copying or recomputation is the most appropriate mechanism. The ECL^{i}PS^e scheduler is dsitributed across the different nodes of the network and based on the concept of message passing.

Finally Parallel ECL^{i}PS^e offers a very flexible worker management. It is possible to add and remove resources during long running applications.

The first release of Parallel ECL^{i}PS^e has been at the beginning of 1995. It provided ECL^{i}PS^e on shared memory machines. First target platforms were the multiprocessors of SUN and the ICL DRS 6000. Parallel ECL^{i}PS^e for distributed memory is currently under development.

The ECLiPSe system has been licensed to around 250 institutions including both industrial and academic sites. More information concerning ECL^{i}PS^e can be found on WWW under the URL: http://www.ecrc.de/eclipse/eclipse.html.

Chapter 2.

A Tutorial on Parallelism and Constraints in ECL^{i}PS^e

Steven Prestwich

1 Introduction

Logic programming has the well-known advantages of ease of programming (because of its high level of abstraction) and clarity (because of its logical semantics). The main drawback is its slow execution times compared to those of conventional, imperative languages. In recent years, research has produced various extensions which make such systems competitive.

ECL^{i}PS^e, the ECRC Prolog platform, is a logic programming system with several extensions. Two of these extensions are targeted at problems with large search spaces; these are *constraint handling* and *parallelism*. Constraints are used to prune search spaces, whereas parallelism exploits parallel or distributed machines to search large spaces more quickly. These complementary techniques can be used separately or combined to obtain clear, concise and efficient programs. These extensions originated in other ECRC systems: constraint handling came from CHIP and the parallelism from ElipSys, both with some changes.

This tutorial is adapted and extended from a similar tutorial for ElipSys [Pre93a]. It provides some general principles on how to make the best use of parallelism and constraints. It is intended as an introduction for the working programmer, and does not contain details of all the built-in predicates available. These details can be found in the Manuals.

1.1 How to read this tutorial

If you are just interested in OR-parallelism then go directly to Section 2, which is self-contained. This is the most important form of parallelism in ECL^{i}PS^e. If you are just interested in AND-parallelism then read Section 2 followed by Section 3, because 2 contains information necessary to understand 3. When you are ready to test OR- or AND- parallel programs for performance, Appendix 5 describes how to handle timing variations when calculating parallel speedups, and includes a note on speedup curves. If you are just interested in constraints then jump to Section 4 which is self contained, except for Section 4.5 which links the ideas of constraints and parallelism. If you are interested in all aspects of parallelism and constraints, then just read the sections in order.

2 OR-Parallelism

Many programming tasks can be naturally split into two or more independent subtasks. If these subtasks can be executed in parallel on different processors, much greater speeds can be achieved. Parallel hardware is becoming cheaper and more widely available, but programming these machines can be much more difficult than programming sequential machines. Using conventional, imperative languages may involve the programmer in a great deal of low-level detail such as communication, load balancing etc. This difficulty in exploiting the hardware is sometimes called the *programming bottleneck*. ECL^{i}PS^{e}avoids this bottleneck. It exploits parallel hardware in an almost transparent way, with the system taking most of the low-level decisions. However, there are still certain programming decisions to be made regarding parallelism, and this tutorial gives some practical hints on

how to make these decisions. In the future even greater transparency will be achieved as analysis and transformation tools are developed.

2.1 How to use it

First, we must tell ECL^{i}PS^e how many processors to allocate for the program. One way to do this is to specify the number of **workers** when parallel ECL^{i}PS^e is called. For example

```
peclipse -w 3
```

calls parallel ECL^{i}PS^e with 3 workers. If no number of workers is specified, ECL^{i}PS^e will simply run sequentially with the default of 1 worker. Other ways of changing the number of workers are described in [ECL95, ECL94].

Note: For the purpose of this tutorial we shall assume that a worker and a processor are the same thing, though there is a subtle difference: it is possible to specify a greater number of workers than there are processors, in which case ECL^{i}PS^e will simulate extra processors. Simulated parallelism is useful for some search algorithms, as it causes the program to search in a more "breadth-first" way. However, it does add an overhead and uses more memory, so it should only be used when necessary.

As a simple example, here is part of a program:

```
p(X) :- p1(X).
p(X) :- p2(X).
```

In a standard Prolog execution, a call to `p` will first enumerate the answers to `p1`, then on backtracking those of `p2`.

We can tell ECL^{i}PS^e to try `p1` and `p2` in parallel instead of by backtracking simply by inserting a declaration

```
:- parallel p/1.

p(X) :- p1(X).
p(X) :- p2(X).
```

The set of answers for `p` will still be the same in parallel as in the backtracking computation, though possibly in a different order.

For convenience, some built-in predicates have been pre-defined as OR-parallel in the library `par_util`. For example `par_member/2` is an OR-parallel version of the list membership predicate `member/2`. Before defining new parallel predicates it is worth checking whether they already exist in the library.

2.2 How it works

The computation of `p` splits into two (or more if there are more `p` clauses) parallel computations which may be executed on separate workers if any are available, and if

ECL^{i}PS^e decides to do so — these decisions are made automatically by the ECL^{i}PS^e task scheduler, and need not concern the programmer.

Continuations

Not only will `p1` and `p2` be computed in parallel, but also any calls occurring after `p` in the computation. This part of the computation is called the **continuation** of the `p` call.

For example, if `p` is called from another predicate:

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

s(X) :- u(X), p(X), v(X).

:- parallel p/1.

p(X) :- p1(X).
p(X) :- p2(X).
```

then `p(X)` has two alternative continuations in a computation of `:- q(f(A),Y)`:

```
p1(f(A)), v(f(A)), t(Y)
p2(f(A)), v(f(A)), t(Y)
```

and it is these processes which will be assigned to separate workers. The idea of a continuation plays a large part in deciding where to use OR-parallelism.

2.3 When to use it

When should we declare predicates as OR-parallel? It may appear that all predicates with more than one clause should be parallel, but this is wrong. In this section we discuss why it is wrong, indicate possible pitfalls, and consider the effects of OR-parallelism on execution times.

Non-deterministic calls

Since a parallel declaration tells the system that the clauses of the predicate should be tried in parallel, clearly only predicates with more than one clause are candidates. Furthermore, deterministic predicates should not be parallel, that is those whose calls only ever match one clause at runtime. For example, the standard list append predicate:

```
append([],A,A).
append([A|B],C,[A|D]) :- append(B,C,D).
```

```
q :- ... p(X) ...

:- parallel p/1.

p(X) :- guard1(X), !.
p(X) :- guard2(X), !.
p(X) :- guard3(X), !.
```

Figure 2.1: *A simple predicate with commit*

is commonly called with its first argument bound to concatenate two lists. Only one clause will match any such call, and there is no point in making `append` parallel. If `append` is called in other modes, for example with only its third argument bound, then it is nondeterministic and may be worth parallelising.

Side effects

Only predicates whose solution order is unimportant should be parallel. An example of a predicate whose solution order may be important is

```
p :- generate1(X), write(X).
p :- generate2(X), write(X).
p :- generate3(X), write(X).
```

where `generate{1,2,3}` generate values for `X` non-deterministically. If `p` is parallelised then the order of `write`'s may change. In fact any side effects in the continuation of a parallel call may occur in a different order. This may or may not be important, only the programmer can decide.

Even if solution order is unimportant, it is recommended that any predicates with side effects such as `read`, `write` or `setval` are only used in sequential parts of the program, otherwise the performance of the system may be degraded. The OR-parallelism of ECL^{i}PS^{e}is really designed to be used for pure search. If parallel solutions are to be collected then there are built-in predicates like `findall` which should be used.

The commit operator

When the cut `!` is used in parallel predicates, it has a slightly different meaning than in normal (sequential) predicates. When used in parallel in ECL^{i}PS^{e}it is called the *commit* operator. Its meaning can be explained using examples.

First consider Figure 2.1. The "guards" execute in parallel, and as soon as one finds a solution the commit operator aborts the other two guards. Only the continuation of the successful guard survives.

This simple example can be written in another way using the `once` meta-predicate, as in Figure 2.2. This is a matter of preferred style.

```
q :- ... once(p(X)) ...

:- parallel p/1.

p(X) :- guard1(X).
p(X) :- guard2(X).
p(X) :- guard3(X).
```

Figure 2.2: *Replacing commits by "once" in a simple parallel predicate*

```
q :- ... p(X) ...

:- parallel p/1.

p(X) :- guard1(X), !, body1(X).
p(X) :- guard2(X), !, body2(X).
p(X) :- guard3(X), !, body3(X).
```

Figure 2.3: *A less simple predicate with commit*

In the more general case there may be calls after the commits, as in Figure 2.3. The commit has exactly the same effect as before, with the `body` calls at the start of the parallel continuations. By the way, this example cannot be rewritten using `once` without a little program restructuring, because of the `body` calls.

Some predicates may have an empty guard, corresponding to (for example) the "else" in Pascal. An example is shown in Figure 2.4 The meaning of this predicate is "if `guard1` then `body1`, else if `guard2` then `body2`, else `body3`". This *must not* be parallelised simply by adding a declaration, because the empty guard may change the meaning of the program when executed in parallel. The reason is that if we make `p` parallel then we may get `body3` succeeding followed by `(guard1, !, body1)` or `(guard2, !, body2)` giving more solutions for `p` than in backtracking mode.

It is safer to use commits in each of the `p` clauses and to introduce a new guard, as in Figure 2.5. This is now safe, but at the expense of introducing extra work (the negated guards in the third clause). A safe and efficient method, though slightly more complicated, is shown in Figure 2.6 where we split the definition of `p` into parallel and sequential parts.

```
p(X) :- guard1(X), !, body1(X).
p(X) :- guard2(X), !, body2(X).
p(X) :-               body3(X).
```

Figure 2.4: *A typical sequential predicate with empty guard*

```
p(X) :-                     guard1(X), !, body1(X).
p(X) :-                     guard2(X), !, body2(X).
p(X) :- not guard1(X), not guard2(X), !, body3(X).
```

Figure 2.5: *Handling empty guards when parallelising*

```
p(X) :- guards(X,N), !, bodies1_and_2(N,X).
p(X) :- body3(X).

:- parallel guards/2.

guards(X,1) :- guard1(X).
guards(X,2) :- guard2(X).

bodies_1_and_2(1,X) :- body1(X).
bodies_1_and_2(2,X) :- body2(X).
```

Figure 2.6: *Another way of handling empty guards when parallelising*

Parallelisation overhead

Even if a program is safely parallelised it may not be worthwhile making a predicate parallel. For example, in a Quick Sort program there is typically a `partition` predicate as shown in Figure 2.7. Although for most calls to `partition` both clauses 2 and 3 will match, one of them will fail almost immediately because of the comparison `H<D` or `H>=D`. There is no point in making `partition` parallel because the overhead of starting a parallel process will greatly outweigh the small advantage of making the comparisons in parallel.

To express the fact that the overhead of spawning parallel processes is equivalent to a significant computation (depending upon the hardware, perhaps as much as several hundred resolution steps) we say that ECL^{i}PS^{e}supports **coarse-grained parallelism**. The **grain size** of a parallel task refers to the cost of its computation, roughly equivalent to its cpu time. Only computations with grain size at least as large as this overhead are worth executing in parallel, in fact the grain size should be much larger than the overhead. Computations which are not coarse-grained are called **fine-grained**.

Estimating grain sizes is usually not as obvious as in the Quick Sort example. In fact it is the most difficult aspect of using OR-parallelism, and we therefore spend some time

```
partition([],_,[],[]).
partition([H|T],D,[H|S],B) :-  H<D, partition(T,D,S,B).
partition([H|T],D,S,[H|B]) :- H>=D, partition(T,D,S,B).
```

Figure 2.7: *Quick Sort partitioning predicate*

```
process_value :-
   value(X),
   process(X).

value(1).
value(2).
  :
value(n).
```

Figure 2.8: *Grain size estimation*

discussing it. In the context of OR-parallelism, a parallel task is a continuation, and so when we refer to the grain size of a parallel predicate call we mean the time taken to execute that call plus its continuation. To illustrate this, consider the program in Figure 2.8, where `process(X)` performs some computation using the value of `X`. Now the question is, should `value` be parallel? The answer depends upon the computations of the various `process` calls since the `value` calls are fine-grained. We now discuss this question in some detail.

Grain size for all solutions

Say that we require all solutions of `process_value`. In a backtracking computation the total time to execute `process_value` is approximately

$$t_1 + \ldots + t_n$$

where $t_i = time(\texttt{process(i)})$. In an OR-parallel computation (assuming sufficient workers are available) the total computation time is approximately

$$k + maximum(t_1 \ldots t_n)$$

where k is the overhead of starting a parallel process, which is machine and implementation dependent. As can be seen from the formulae, if `process` has

- *no* expensive calls then k becomes significant, and the backtracking computation is faster;
- *one* expensive call then the sequential and parallel cases will take about the same time;
- *two or more* expensive calls then `k` is insignificant and the parallel computation is faster.

The programmer must try to ensure that at least two continuations have significant cost.

```
one_process_value :-
   value(X),
   process1(X),
   !,
   process2(X).

:- parallel value/1.

value(1).
value(2).
  :
value(n).
```

Figure 2.9: *Grain size estimation and an obvious commit*

Grain size for one solution

Say that we only require `process_value` to succeed once. In a backtracking computation the time will be

$$t_1 + \ldots + t_s$$

where s is the number of the first succeeding `process(s)`. In a parallel computation the time will be

$$k + minimum(t_1 \ldots t_n)$$

Now the parallel computation is only cheaper if there are *one or more* values of $i \leq s$ for which `process(i)` is expensive.

Grain size and pruning operators

Pruning operators such as the commit may affect estimates of the grain size of a continuation. Consider the program in Figure 2.9. Here `n` processes will be spawned with continuations

```
process1(1), !, process2(1).
:
process1(n), !, process2(n).
```

As soon as `process1` on one worker succeeds, all the other workers will abandon their computations. Hence the actual grain size of any continuation of an OR-parallel call is no greater than that of the cheapest process before pruning occurs. Of course, it may be smaller than this if failure occurs before the pruning operator is reached.

In fact, we must consider the effects of commits in *any* predicate which calls a parallel predicate, even indirectly. For example, see the program in Figure 2.10. Since `p1` contains a commit which prunes `p2`, and `p2` calls `value` (indirectly), we only need to estimate the grain size of the continuation up to the commit, that is the grain size of

```
p1 :- p2, !.

p2 :- process_value, p3.

process_value :-
   value(X),
   process(X),
```

Figure 2.10: *Grain size estimation and a less obvious commit*

```
delay expensive_process(A) if nonground(A).

p :- expensive_process(X), process, X=0.

process :- cheap_process1.
process :- cheap_process2.
```

Figure 2.11: *Grain size estimation and coroutining, first example*

```
process(X), p3
```

The same holds for any pruning operator, including `once/1`, `not/1` and `->` (if-then-else) because these contain implicit commits. When we talk about a continuation for an OR-parallel call in future, we shall mean the continuation up to the first pruning operator.

Grain size and coroutining

When estimating the grain size of a continuation, we must take into account any suspended calls which may be woken during the computation. For example, consider the program in Figure 2.11. When deciding whether to parallelise `process` we estimate the grain sizes of

```
cheap_process1, X=0
cheap_process2, X=0
```

These *appear* to be cheap, but at runtime `X=0` wakes `expensive_process` and so it is effectively expensive.

On the other hand, given the program in Figure 2.12, it appears that `process` has two expensive continuations

```
cheap_process1, expensive_process(X)
cheap_process2, expensive_process(X)
```

before the commit occurs, but this is deceptive because `expensive_process` is not woken until after the commit.

```
delay expensive_process(A) if nonground(A).

p :- process, expensive_process(X), !, X=0.

process :- cheap_process1.
process :- cheap_process2.
```

Figure 2.12: *Grain size estimation and coroutining, second example*

```
berghel :-
   word(A1,A2,A3,A4,A5),   % column 1
   word(A1,B1,C1,D1,E1),   %    row 1
   word(B1,B2,B3,B4,B5),   % column 2
   word(A2,B2,C2,D2,E2),   %    row 2
   word(C1,C2,C3,C4,C5),   % column 3
   word(A3,B3,C3,D3,E3),   %    row 3
   word(D1,D2,D3,D4,D5),   % column 4
   word(A4,B4,C4,D4,E4),   %    row 4
   word(E1,E2,E3,E4,E5),   % column 5
   word(A5,B5,C5,D5,E5).   %    row 5

word(a,a,r,o,n).
word(a,b,a,s,e).
word(a,b,b,a,s).
  :
```

Figure 2.13: *Sequential Berghel program*

Parallelisation of predicates

So far we have discussed when it is worthwhile making a *call* OR-parallel. However, in ECL^{i}PS^e we parallelise calls indirectly by deciding whether to declare a *predicate* parallel or not. To do this, the programmer must consider the most important calls to the predicate, that is the calls which have greatest effect on the total computation. If they would be faster in parallel then the predicate should be declared as parallel. For some predicates this may be easy to see but others may be called in many different ways.

For example consider the Berghel problem. We are given a dictionary of 134 words each with 5 letters. We must choose 10 of them which can be placed in a 5×5 grid. The program is shown in Figure 2.13. Is it worth making `word` parallel?

We must consider grain sizes. During the computation of `berghel` there will be many calls to `word`, with all, some or none of the arguments bound to a letter. The grain size will depend partly upon how many letters are bound. It will also depend upon the bound letters themselves, for example binding an argument to a `z` will almost certainly prune the search more than binding it to an `a`. Another factor is the continuation of each call.

The continuation of the fifth call is

```
word(A3,B3,C3,D3,E3),
word(D1,D2,D3,D4,D5),
word(A4,B4,C4,D4,E4),
word(E1,E2,E3,E4,E5),
word(A5,B5,C5,D5,E5)
```

whereas that of the eighth call is only

```
word(E1,E2,E3,E4,E5),
word(A5,B5,C5,D5,E5)
```

The cheaper calls may be slower when called in parallel and the more expensive calls faster.

The result of parallelising `word` is the net result of all these effects, which can best be estimated by experimentation (trace visualisation and profiling tools, when available, can help greatly).

Parallelisation of calls

We can make more selective use of OR-parallelism by parallelising only *some* calls. In the Berghel example, if we keep `word` sequential and add a new parallel version as in Figure 2.14 then we can experiment by replacing various calls to `word` by calls to `parword`. The question is, which calls should be parallel?

Running this program on a SUN SPARCstation 10 model 51 with 4 CPU's it turns out that the best result (a speedup of about 3.3) is obtained when *all* the calls are parallel — in other words, simply declaring `word` parallel. However, this may not be true for all machines and all numbers of workers. This example behaves differently in experiments with ElipSys on a Sequent Symmetry with 10 workers, and we conjecture that similar effects will be observed in ECL^{i}PS^e with more workers, or on parallel machines with faster cpu's. With all `word` calls parallel we get a speedup of 6.7, but if we only parallelise the first 2 calls as in Figure 2.14 we obtain an almost linear speedup of 9.7. So in some cases it is worth a little experimentation and programming effort to selectively parallelise calls.

In this example we chose between the parallel and sequential versions according to a static test: the position of the call in a clause. The choice could also be based on a dynamic property such as instantiation patterns.

Partial parallelisation

Recall that it is worth parallelising a predicate if (for most of its calls) there are at least two clauses leading to large-grained continuations. If we can predict *which* of the clauses may lead to such continuations then we can extract them from the predicate definition, and avoid spawning small-grained parallel processes.

```
berghel :-
   parword(A1,A2,A3,A4,A5),
   parword(A1,B1,C1,D1,E1),
   word(B1,B2,B3,B4,B5),
   word(A2,B2,C2,D2,E2),
   word(C1,C2,C3,C4,C5),
   word(A3,B3,C3,D3,E3),
   word(D1,D2,D3,D4,D5),
   word(A4,B4,C4,D4,E4),
   word(E1,E2,E3,E4,E5),
   word(A5,B5,C5,D5,E5).

:- parallel parword/5.

parword(a,a,r,o,n).
parword(a,b,a,s,e).
parword(a,b,b,a,s).
  :
```

Figure 2.14: *Parallelising selected calls*

```
process_value :-
   value(X),
   process(X).

value(1). % leads to cheap process
value(2). % leads to expensive process
value(3). % leads to cheap process
value(4). % leads to expensive process
```

Figure 2.15: *A program worth partially parallelising*

```
process_value :-
   value(X),
   process(X).

value(X) :- value13(X).
value(X) :- value24(X).

value13(1).
value13(3).

:- parallel value24/1.

value24(2).
value24(4).
```

Figure 2.16: *Partially parallelised version*

```
p(X) :- q(X), new(X), r(X).

:- parallel new/1.

new(X) :- a(X).
new(X) :- b(X).
```

Figure 2.17: *Parallelised disjunction*

For example, consider the sequential program in Figure 2.15. If we know that `process(i)` is small-grained for `i` $= \{1, 3\}$ but large-grained for `i` $= \{2, 4\}$ then it is best to decompose `value` into backtracking and parallel parts, as shown in the parallel program of Figure 2.16. Then values $1, 3$ are handled by backtracking while values $2, 4$ are handled in parallel.

Parallelisation of disjunctions

ECL^{i}PS^e(in common with most Prolog dialects) allows the use of disjunctions in a clause body. For example,

```
p(X) :- q(X), (a(X); b(X)), r(X).
```

It may be worthwhile calling `a` and `b` in OR-parallel mode if `a, r` and `b, r` (plus any continuation of `p`) have sufficiently large grain. The use of disjunction is really a notational convenience, and may hide potential parallelism. Of course it would be possible to add a parallel disjunction operator to ECL^{i}PS^e, but this is unnecessary because we can instead make a new, parallel definition as shown in Figure 2.17.

```
:- worker(2).

p(X) :- ascending(X),
p(X) :- descending(X).
```

Figure 2.18: *Ascending-descending example*

Speedup

Assuming we have OR-parallelised a program well, what speedup can we expect? The answer depends on whether we want all solutions to a call or just one.

Speedup for all solutions When parallelising a predicate, we often hope for *linear speedup*. That is, if we have N workers then we want queries to run N times faster. Because of the overhead of spawning parallel processes we usually obtain sublinear speedup, though with fine tuning we may approach linearity.

Consider the program shown in Figure 2.18 where `ascending(X)` has answers

```
X=1, X=2, ... X=1000
```

`descending(X)` has answers

```
X=1000, ... X=2, X=1
```

and both `ascending(X)` and `descending(X)` take time t to find each successive answer (where t is much greater than the parallel overhead k).

With 2 workers the time taken to find all solutions for `p` is $2000t$ with `p` sequential, but $1000t + k$ with `p` parallel: almost linear speedup.

Speedup for one solution However, when using a predicate to find *one* solution, we generally find little relationship between execution times in backtracking and OR-parallel modes, except when averaged over many queries. This is because solutions may not be distributed evenly over the search space.

The time to find one solution for the query `p(X), X=1000` is $1000t$ with `p` sequential (999 failing calls followed by 1 succeeding call to `ascending`), but $t + k$ with `p` parallel (an immediately succeeding call to `descending`). This is a speedup of almost 1000 using only 2 workers: very superlinear speedup.

On the other hand, the time to find one solution for the query `p(X), X=1` is t with `p` sequential and $t + k$ with `p` parallel: no speedup at all.

This shows that for single-solution queries the difference between superlinear speedup and no speedup may depend only on the query.

2.4 Summary

The best use will be made of OR-parallelism if the programmer keeps it in mind from the start. However, a program written for sequential ECL^{i}PS^e can be parallelised using the principles outlined in this section. Here is a summary of the principles.

- Look for predicates which are worth declaring as OR-parallel. When deciding this, all runtime calls to the predicate must be considered. If all, or almost all, calls to a predicate would be faster in OR-parallel, and if it is always safe to do so, then it is worth declaring the predicate as parallel. If it is sometimes worth calling in OR-parallel and sometimes not (but always safe), then a useful technique is to make a parallel and a sequential definition of the predicate and use them where appropriate.
- A call is unsafe in OR-parallel if it has side effects in any of its continuations, or if it has commits in some but not all of its clauses.
- A call is (probably) faster in OR-parallel if it has at least two expensive continuations. A continuation should only be considered up to the first commit or other pruning operator which affects it, and taking into account any suspended calls.
- To further refine a program, look for parallel predicates with some clauses which do not have expensive continuations, then isolate the useful clauses in a new parallel definition. Also look for disjunctions in clause bodies which may hide parallelism, and replace these by calls to new parallel predicates.
- The `once` operator is sometimes stylistically preferable to the use of commits in parallel predicates.

However, these principles do not guarantee the best speedups. In [Pre93b, Pre94c] we described various ways in which (for example) two parallel declarations could combine to give a poor speedup, even though each alone gave a good speedup. We also showed that improving a parallel predicate may have a good, bad or no effect on overall speedup. Effects like these make tuning a parallel program rather harder than tuning a sequential one. Note that they are not ECL^{i}PS^e bugs and will occur in many parallel programs. They may be more obvious in ECL^{i}PS^e since parallelisation of logic programs is very easy. The significance of these effects is that they make it hard to recommend a good general strategy. Probably the best approach is common sense based on knowledge of the program, plus the use of available programming tools. ECL^{i}PS^e will soon have at least one trace visualisation tool to aid parallelisation.

3 Independent AND-parallelism

As well as OR-parallelism ECL^{i}PS^e supports independent AND-parallelism, which is used in quite different circumstances. AND-parallelism replaces the left-to-right computation rule of Prolog by calling two or more calls in parallel and then collecting the results. *Dependent* AND-parallelism is rather different, and is outside the scope of this tutorial.

```
p(X) :- q(X), r(X).

q(a).           r(c).
q(b).           r(d).
q(c).           r(e).
```

Figure 3.1: *Simple AND-parallelism example*

3.1 How to use it

As with OR-parallelism, we need to tell ECL^{i}PS^e how many workers to allocate. Then we simply replace the usual "`,`" conjunction operator by a parallel operator "`&`"; for example replace

```
p(X) :- q(X), r(X).
```

by

```
p(X) :- q(X) & r(X).
```

More than two calls can be connected by `&`.

For convenience there is a built-in predicate which can be used to map one list to another. This is `maplist`, and it applies a specified predicate to each member in AND-parallel. See [ECL95, ECL94] for details.

3.2 How it works

As an example (which is not to be taken as a useful candidate for AND-parallelism, but only as an illustration), consider the program in Figure 3.1. In standard Prolog, given a query `:-p(X)`, `q` is first solved to return the answer `X=a` then `r` is called, fails, and backtracking occurs. The next solution to `q` is `X=b` and again `r` fails. For the next solution `X=c`, `r` succeeds. On backtracking no more solutions are found.

Now if we call `q` and `r` in AND-parallel:

```
p(X) :- q(X) & r(X).
```

what happens instead is that the solutions `{X=a, X=b, X=c}` of `q` and `{X=c, X=d, X=e}` of `r` are collected independently using different workers, and then the results are merged to give the consistent set `{X=c}`. This is clearly a rather different strategy for executing a program, and in this section we discuss when it is better than the usual strategy.

As with OR-parallelism, it is not always true that different workers will be assigned to AND-parallel calls, depending upon runtime availability. This need not concern the programmer.

3.3 When to use it

When should AND-parallelism be used? It may seem at first glance that it will always be faster than the usual sequential strategy, but as often with parallelism this intuition is wrong. In this section we discuss when to apply AND-parallelism.

Non-logical calls

It is sometimes *incorrect* to use AND-parallelism because of side effects and other non-logical Prolog features. For example

```
p(X) :- generate(X), test(X).

test(X) :- X\==badterm, rest_of_test(X).
```

Here `generate(X)` binds `X` to some term, and `test(X)` performs some test on `X`, including the non-logical test `X\==badterm`. Say that the answers to `generate(X)` are

```
{X=goodterm1, X=goodterm2, X=badterm}
```

and the terms permitted by `rest_of_test(X)` are

```
{X=goodterm1, X=badterm}
```

Then `p` has only one answer `{X=goodterm1}`

However, if we use AND-parallelism:

```
p(X) :- generate(X) & test(X).
```

then `test(X)` is first called with `X` unbound, and has answers

```
{X=goodterm1, X=badterm}
```

Merging this with the answers for `generate(X)` we get more answers:

```
{X=goodterm1, X=badterm}
```

which is incorrect. Examples can also be found where a program fails instead of generating solutions.

Non-terminating calls

AND-parallel calls must terminate when called in any order. For example, given

```
p(L1,L2) :-
   append([LeftHead|LeftTail],Right,L1),
   append(Right,[LeftHead|LeftTail],L2).
```

where `append` is the usual list append predicate. This program with a query

```
:-p([1,2,3],L2)
```

would give answers

```
{L2=[2,3,1], L2=[3,1,2], L2=[1,2,3]}
```

But if we use AND-parallelism:

```
p(L1,L2) :-
   append([LeftHead|LeftTail],Right,L1) &
   append(Right,[LeftHead|LeftTail],L2).
```

then the call

```
append(Right,[LeftHead|LeftTail],L2)
```

will not terminate because `Right` is unbound.

Shared variables

Even if the calls can safely be executed in any order, it is not necessarily worth calling them in AND-parallel. If the answers to one call restrict the answers to another call, then this pruning effect may give greater speed than finding all the answers to both calls and then merging the results.

For example consider

```
p(X) :- compute1(X), compute2(X).

compute2(X) :- cheap_filter(X), compute3(X).
```

where `compute1(X)` has the answers

```
{X=1, X=2, ... X=1000}
```

and `cheap_filter(X)` allows the bindings

```
quicksort([Discriminant|List],Sorted) :-
   partition(List,Discriminant,Smaller,Bigger),
   quicksort(Smaller,SortedSmaller),
   quicksort(Bigger,SortedBigger),
   append(SortedSmaller,[Discriminant|SortedBigger],Sorted).
```

Figure 3.2: *Sequential Quick Sort program*

```
{X=1000, X=1001, ... X=1999}
```

Say `compute3` performs some expensive computation on `X`. Now given a query `:-p(X)`, `X` is generated by `compute1(X)` and `cheap_filter` quickly rejects all answers except `X=1000`, so that `compute3(X)` is only called once. The total computation time for all solutions is (ignoring the times of `cheap_filter` for simplicity)

$$time(\texttt{compute1(1)}) + \ldots + time(\texttt{compute1(1000)}) + time(\texttt{compute3(1000)})$$

If we use AND-parallelism instead:

```
p(X) :- compute1(X) & compute2(X).
```

then `compute2(X)` is called with `X` unbound and `compute3(X)` is called 1000 times for each permitted answer of `cheap_filter(X)`. The total computation time for all solutions is now (ignoring the parallelism overhead)

$$maximum(time(\texttt{compute1(1)}) + \ldots + time(\texttt{compute1(1000)}), time(\texttt{compute3(1000)}) + \ldots + time(\texttt{compute3(1999)}))$$

Comparing the two times, it can be seen that the parallel time will be slower than the sequential time if `compute3` is more expensive than `compute1`. By calling `compute1` and `compute2` independently we lose the pruning effect of `compute1` on `compute2`. In fact, in this example `cheap_filter` should not be used in independent AND-parallel, but as a constraint or a delayed goal.

Parallelisation overhead

As with OR-parallelism, we must consider the overhead of creating parallel processes, and only parallelise calls with large grain size. When estimating grain size for AND-parallelism we do not need to consider continuations, only the grain size of the calls themselves. Also, because of the way AND-parallelism is implemented we always estimate grain size for all solutions, never for one solution.

Consider the Quick Sort program in Figure 3.2. For large lists `Smaller` and `Bigger` the grain sizes of the recursive `quicksort` calls may be large enough to justify calling them in parallel, as in Figure 3.3. Of course, as the input list is decomposed into smaller and smaller sublists parallelisation becomes less worthwhile.

```
quicksort([Discriminant|List],Sorted) :-
   partition(List,Discriminant,Smaller,Bigger),
   quicksort(Smaller,SortedSmaller) &
   quicksort(Bigger,SortedBigger),
   append(SortedSmaller,[Discriminant|SortedBigger],Sorted).
```

Figure 3.3: *AND-parallel Quick Sort program*

```
quicksort([Discriminant|List],Sorted) :-
   partition(List,Discriminant,Smaller,Bigger),
   length(Smaller,SmallerLength),
   length(Bigger,BiggerLength),
   (SmallerLength>30,
    BiggerLength>30 ->
      quicksort(Smaller,SortedSmaller) &
      quicksort(Bigger,SortedBigger)
   ;  quicksort(Smaller,SortedSmaller),
      quicksort(Bigger,SortedBigger)),
   append(SortedSmaller,[Discriminant|SortedBigger],Sorted).
```

Figure 3.4: *Conditional AND-parallel Quick Sort program*

In fact Quick Sort is not a good example for ECL^{i}PS^e because it is more concerned with OR-parallelism, and its implementation of AND-parallelism is not very sophisticated. Since it collects all the results of two AND-parallel goals, there is an overhead which grows as the sizes of the goal arguments grow. For the Quick Sort program, coarse-grained goals also have large terms, and so it is probably never worthwhile using AND-parallelism. We shall use Quick Sort for purposes of illustration and pretend that this overhead does not exist, but the reader should be aware that goals should only be called in AND-parallel when their arguments are not very large.

Conditional parallelisation

We can make more efficient use of AND-parallelism by introducing runtime tests. Say that for a given number of workers, lists with length greater than 30 make parallelisation worthwhile, while smaller lists cause fine-grained recursive calls which do not make it worthwhile. Then we can write the program shown in Figure 3.4.

This can be further refined by making `partition` calculate the lengths of `Smaller` and `Bigger` as they are constructed, to avoid the expensive calls to `length`. In fact, we should be careful of introducing expensive runtime tests.

A point worth noting is that when estimating the grain size of a `quicksort(L)` call to set the threshold (30 in this case) we should base the estimate on the version *with* the runtime test. The version with the tests will have greater grain size for a given list length,

and so the threshold can be set lower, giving greater parallelism.

Speedup

It is possible to obtain superlinear speedup with AND-parallelism. For example, say we have AND-parallel calls `(a & b)` where `b` fails immediately. Then `a` can be aborted immediately. But if instead we had called `(a, b)` the failure of `b` would not be detected until after `a` had completed, thus AND-parallelism may cause a large speedup. [1]

However, if none of the AND-parallel calls fails then the expected speedup is linear or sublinear. Unlike OR-parallelism all solutions of AND-parallel calls are computed, and so there is no difference between one-solution and all-solution queries. However, when there are not enough workers available AND-parallel calls will be called using the same worker, as already mentioned. This execution will be noticeably less efficient than a normal sequential execution. Therefore AND-parallel calls need to have large grain size so that the overhead is not significant.

3.4 Summary

A program written for sequential ECL^{i}PS^e can be AND-parallelised using the principles outlined in this section. Here is a summary of the principles.

- Look for conjunctions of calls which can be called in AND-parallel. First consider whether they are safe in parallel. It is unsafe to AND-parallelise calls sharing variables which are used in non-logical calls such as `var(X)`, `X\==Y`, `setval(X,Y)` and `read(X)`. It is also unsafe to AND-parallelise calls whose results depend upon the order in which they are called.
- Next consider whether they will be faster in parallel than in sequence. Only expensive calls with small arguments are worth calling in parallel. Also, calls which compete to bind some shared variable will probably be faster when called sequentially. If a cheap way can be found to estimate the grain sizes of calls at runtime, then this can be used in a runtime test to choose between sequential and AND-parallel execution.

As with OR-parallelism, there is no strategy which always leads to the best speedups. However, a common-sense approach works well in most cases.

4 Finite Domain constraint handling

Constraint handling can speed up search problems by several orders of magnitude, by pruning the search space in the *forward* direction (*a priori*), in contrast to backtracking search which prunes in the *backward* direction (*a posteriori*). Many difficult discrete

[1] However, at the time of writing ECL^{i}PS^e will *not* detect the failure of `b` in this example; it may in future versions.

combinatorial problems can be solved using constraints which are beyond the reach of pure logic programming systems. Such problems can of course be solved by special purpose programs written in imperative languages such as Fortran, but this involves a great deal of work and results in large programs which are hard to modify or extend. CLP programs are much smaller, clearer and easier to experiment with. ECL^{i}PS^{e}has incorporated a number of constraint handling facilities for this purpose. For an overview on constraint logic programming see [vH89], from which some of the examples below have been adapted.

We shall illustrate how to use the finite domains in ECL^{i}PS^{e}with a single example: the overused but useful 8-queens problem.

4.1 Description of the 8-queens problem

Consider a typical combinatorial problem. We have several variables each of which can take values from some finite domain. Choosing a value for any variable imposes restrictions on the other variables. The problem is to find a consistent set of values for all the variables.

For example, consider the ubiquitous 8-queens problem. We have a chess board, 8×8 squares, and 8 queens, and we wish (for some reason) to place all these queens on the board so that no queen attacks another. It is well known that there are 92 ways of doing this.

Placing any queen on the board typically imposes new restrictions by attacking several new squares: along the vertical, horizontal and two diagonal lines. It is possible to imagine many strategies for placing the queens on the board. We now discuss some of these and their expression in ECL^{i}PS^e.

4.2 Logic programming methods

Before describing how to use constraints, we give several versions without constraints. These will help to illustrate the later versions and to contrast the two approaches.

Generate-and-test

The most obvious formulation is a purely generate-and-test program which places all the queens on the board and then checks for consistency (no queen attacks another). This is shown in Figure 4.1: `permutation` is a library predicate which generates every possible permutation of the list `[1,2,3,4,5,6,7,8]` non-deterministically, and `safe` checks for consistency. The first number in the list denotes the row of the first queen (in column 1), the second number the row of the second queen (in column 2) and so on.

This is arguably the most natural program, but extremely inefficient.

```
eight_queens(Columns) :-
   Columns = [_,_,_,_,_,_,_,_],
   Numbers = [1,2,3,4,5,6,7,8],
   permutation(Columns,Numbers),
   safe(Columns).

safe([]).
safe([Column|Columns]) :-
   noattack(Column,Columns,1),
   safe(Columns).

noattack(Column,[],Offset).
noattack(Column,[Number|Numbers],Offset) :-
   Column =\= Number - Offset,
   Column =\= Number + Offset,
   NewOffset is Offset + 1,
   noattack(Column,Numbers,NewOffset).
```

Figure 4.1: *8-queens by generate-and-test*

Test-and-generate

With a small change, the generate-and-test program can be made quite good. We simply reverse the calls in the `eight_queens` clause and use coroutining to suspend the checks until they can be made. This is shown in Figure 4.2. Now all the checks are set up initially and suspended, and then the queens are placed one by one. Each time a queen is placed the relevant checks are woken immediately, thus interleaving placements with checks. This is closer to the way in which a human would proceed.

Standard backtracking

The next most obvious formulation is to explicitly interleave the consistency checks with the placing of the queens. A typical such program is shown in Figure 4.3.

This is a fairly clear program, and more efficient than the previous program because it has no coroutining overhead. But it is not the best available; in fact if we increase the number of queens (and the size of the board) it becomes hopelessly inefficient.

Forward checking

The strategy can be improved by a technique called *forward checking*. Each time we place a queen, we immediately remove all attacked squares from the domains of the remaining unplaced queens. The trick is that if any domain becomes empty we can immediately backtrack, whereas in the previous program we would not backtrack until we tried to place the later queen. All the useless steps in between are thus eliminated.

```
eight_queens(Columns) :-
   Columns = [_,_,_,_,_,_,_,_],
   Numbers = [1,2,3,4,5,6,7,8],
   safe(Columns),
   permutation(Columns,Numbers).

noattack(Column,[],Offset).
noattack(Column,[Number|Numbers],Offset) :-
   check(Column,Number,Offset),
   NewOffset is Offset + 1,
   noattack(Column,Numbers,NewOffset).

delay check(A,B,C) if nonground(A).
delay check(A,B,C) if nonground(B).
delay check(A,B,C) if nonground(C).

check(Column,Number,Offset) :-
   Column =\= Number - Offset,
   Column =\= Number + Offset,
```

Figure 4.2: *8-queens by test-and-generate*

```
eight_queens(Columns) :-
   solve(Columns,[],[1,2,3,4,5,6,7,8]).

solve([],_,[]).
solve([Column|Columns],Placed,Number) :-
   delete(Column,Number,Number1),
   noattack(Column,Placed,1),
   solve(Columns,[Column|Placed],Number1).
```

Figure 4.3: *8-queens by standard backtracking*

```
eight_queens(Columns) :-
   Columns=[_,_,_,_,_,_,_,_],
   Columns :: 1 .. 8,
   solve(Columns).

solve([Column]) :-
   indomain(Column).
solve([Column1,Column2|Columns]) :-
   indomain(Column1),
   noattack(Column1,[Column2|Columns],1),
   solve([Column2|Columns]).

noattack(Column,[],Offset).
noattack(Column1,[Column2|Columns],Offset) :-
   Column1 ## Column2,
   Column1 ## Column2 + Offset,
   Column1 ## Column2 - Offset,
   NewOffset is Offset+1,
   noattack(Column1,Columns,NewOffset).
```

Figure 4.4: *8-queens by forward checking*

A Prolog program using forward checking can be written, but we shall not show it here because it is rather long. It maintains a list of possible squares for each queen, and every time a queen is placed these lists must be reduced.

The program is indeed more efficient for a large number (larger than about 12) of queens, but for fewer queens it is less efficient because of the overhead of explicitly handling the variable domains. It is also considerably less clear than the previous program.

4.3 Constraint logic programming methods

We now come to constraint handling. We shall compare and contrast these methods with the Prolog methods described above.

Forward checking

We can very easily write a forward checking program for 8-queens, as in Figure 4.4. The `##` built-in predicate is the ECL^{i}PS^e disequality constraint.

This program looks similar to the standard backtracking program, but even simpler because the variable domains are not explicitly manipulated. Instead they are an implicit property of the domain variables, set up by the call `Columns :: 1 .. 8`. The program works in much the same way as the Prolog forward checking program, but is more efficient.

```
eight_queens(Columns) :-
   Columns=[_,_,_,_,_,_,_,_],
   Columns :: 1 .. 8,
   safe(Columns),
   placequeens(Columns).

safe([]).
safe([Column|Columns]) :-
   noattack(Column,Columns,1),
   safe(Columns).

placequeens([]).
placequeens([Column|Columns]) :-
   indomain(Column),
   placequeens(Columns).
```

Figure 4.5: *8-queens with improved forward checking*

Improved forward checking

We can also write a constraints analogue to the test-and-generate program, which gives more sophisticated forward checking. When we place a queen, not only can we check for empty domains but also for singleton domains. Placing a queen may reduce a remaining queen's domain to one value, and we can immediately place that queen and do further forward checking. This is sometimes called *unit propagation*.

This will be better than the previous program. We should do as much propagation as possible at each step, because a propagation step is deterministic whereas placing a queen is non-deterministic.

The forward checking program provides no opportunity to do this, because when placing each queen not all the constraints have been called yet. We need a different formulation, as in Figure 4.5. This is similar to the test-and-generate program, though much faster because of forward checking. It sets up all the relevant constraints and only then does it begin to place the queens. Note that the `placequeens` call could actually be replaced by a call to the equivalent library predicate `labeling`. However, we will modify `placequeens` below, so it is useful to show it here.

The first-fail principle

Forward checking can be improved by the *first-fail principle*. In this technique, we do not simply place the queens in the arbitrary order `1,2,3...` but instead choose a more intelligent order.

The first-fail principle is a well-known principle in Operations Research, which states that given a set of possible choices we should choose the most deterministic one first. That is, if we have to choose between placing the seventh queen which has 3 possible positions,

```
placequeens([]).
placequeens([Column|Columns]) :-
   deleteff(Column,[Column|Columns],Rest),
   indomain(Column),
   placequeens(Rest).
```

Figure 4.6: *8-queens with improved forward checking plus first-fail*

and the sixth queen which has 5 possible positions, we should place the seventh queen first. We have already seen a limited version of this principle when we selected queens with 0 or 1 possible places first in the forward checking programs.

It is very simple to implement the principle in ECL^{i}PS^e, as shown in Figure 4.6. The `deleteff` built-in deletes the domain variable with the smallest domain from the list of remaining domain variables. Variations on `deleteff` are listed in the Extensions User Manual [ECL94].

Note that it is quite simple to obtain a radically different computation strategy by controlling the way in which variables take domain values. It would be far more difficult to write these strategies directly in a logic program.

Maximising propagation

There is another useful principle which makes a significant improvement to the 8-queens problem. Like the first-fail principle this is concerned with choosing an intelligent order for placing the queens, but the aim here is to perform as much propagation as early as possible.

If we begin by placing the first queen, that is the queen on the first column, this enables ECL^{i}PS^{e}to delete squares from the domains of all the future queens. However, if we begin by placing, say, the fourth queen then *more* squares can be deleted. This is because the middle squares can attack more squares than those on the edges of the board.

4.4 Non-logical calls

With such sophisticated execution strategies it is hard to predict when domain variables will become bound. In this way, constraints are similar to suspended calls (that is, those used in coroutining). If variables become bound at unexpected points in the computation, cuts, side effects and other non-logical built-ins (such as `var` and `\==`) may not have the expected effects. It is therefore advisable to use constraints only in "pure" parts of a program.

```
placequeens([]).
placequeens([Column|Columns]) :-
   deleteff(Column,[Column|Columns],Rest),
   par_indomain(Column),
   placequeens(Rest).
```

Figure 4.7: *8-queens with parallel forward checking plus first-fail*

4.5 Parallelism

The two features of OR-parallelism and constraint handling can easily be combined to yield very efficient and clear programs. Predicates in a CLP program can be parallelised as in a logic program, exactly as described in Sections 2 and 3, and subject to the same restrictions plus those described in Section 4.4.

There is also a more direct interaction between constraints and OR-parallelism. Constraint handling aims to reduce the number of non-deterministic choices in a computation, but such choices must still be made. They can be made in parallel by using a parallel counterpart of `indomain` called `par_indomain` (this is available in the `fd` library).

Any of the previous programs can be parallelised in this way. For example, Figure 4.7 shows a parallel forward checking program.

4.6 Summary

Programs should be written with constraints in mind from the start, because they use a different data representation than logic programs (which do not have domain variables). Here is a summary of the general principles discussed in this section.

- Given a problem, look for ways of using forward checking as opposed to backtracking search, then formulate the forward checking in terms of constraints.
- Try to enhance forward checking by setting up as many constraints as possible before choosing values by `indomain`.
- Try to further reduce backtracking first by choosing values from small domains, and then by choosing values in an order which maximises propagation.
- Beware of using constraints in parts of a logic program with cuts, side-effects or other non-logical features.
- Parallelise CLP programs exactly as with logic programs, also replacing `indomain` by `par_indomain` where it is safe and profitable to do so.

5 Appendix: Calculating parallel speedup

A figure which must often be calculated to evaluate a parallel program is the *parallel speedup*. However, variations in parallel execution times make speedup tricky to measure. In a previous technical report [Pre93b] we described various ways in which a program may give very different execution times when run several times under identical circumstances. This is not a bug of ECL^{i}PS^e but a feature of many parallel programs.

Several ways to cope with these variations can easily be thought of: do we take the mean of the parallel times and then calculate speedup, or do we divide by each parallel time and then take the mean speedup? What sort of mean should we use (arithmetic, geometric, median)?

In a recent paper by Ertel [Ert94] it is shown that, given a few common-sense assumptions about the properties of speedup, there is only *one* sensible way of calculating speedup from varying times. The paper gives quite general results, and this note extracts the details relevant to ECL^{i}PS^e users.

5.1 The obvious definition

Speedup is commonly defined as $S = \frac{T_s}{T_p}$ where T_s is the sequential and T_p the parallel execution time. Because of variations in the parallel system, we may have several parallel times $T_p^1 \ldots T_p^n$ for exactly the same query. For certain types of program (especially single-solution queries) these times may vary wildly. This is not a fault in ECL^{i}PS^e but a feature of certain types of parallelism. The causes are not relevant here, but the effects are. How do we calculate the speedup when parallel times vary?

The usual method is to take the arithmetic mean of the parallel times and then divide to get S. This method has been widely used for years by empirical and theoretical scientists [Ert94], and is appropriate in some cases. A *system designer* who wants to compare the parallel and sequential performance is interested in the reduction of cost in the long run — he wants to compare the sum of many parallel run times with the sum of many sequential run times. Ertel calls this the "designer speedup".

However, the designer speedup is not appropriate for the *user* . A user is interested in the speedup for a single run, and therefore needs the expectation of the ratio $\frac{T_s}{T_p^i}$. Moreover the designer speedup carries no information about the variation of speedup. What is a good definition for "user speedup", and how could we define speedup variation?

To illustrate the problem, say we have

$$T_s = 10 \;\; T_p^1 = 2 \;\; T_p^2 = 50$$

If we take the arithmetic mean of T_p^1 and T_p^2 then calculate the speedup we get $S = 0.38$. If we calculate the two possible speedups and take the arithmetic mean we get $S = 2.6$. If we take the geometric mean in either case we get $S = 1$. Which, if any, is correct?

5.2 A better definition

It is shown in [Ert94] that the correct way to calculate ("user") speedup in these cases is to take the *geometric mean of all possible speedups*. That is given $\{T_s^1 \ldots T_s^n\}$ and $\{T_p^1 \ldots T_p^m\}$ (normally $n = 1$) to take the geometric mean of the ratios

$$\left\{ \frac{T_s^i}{T_p^j} \;\middle|\; i = 1 \ldots n,\, j = 1 \ldots m \right\}$$

In the example above this gives $S = 1$, which is sensible: using T_p^1 we have $S = 5$ and using T_p^2 we have $S = \frac{1}{5}$, so "on average" we get the product $S = 1$. For technical reasons on *why* this is the correct method, see [Ert94].

A note on calculation

The geometric mean of n numbers is the n^{th} root of their product. If the numbers are too large to multiply together, this can be calculated by taking the arithmetic mean a of their natural logarithms, and then calculating e^a.

If some of the parallel times or speedups are identical, they must still be treated as if they are different. For example, if

$$T_s = 10 \;\; T_p^1 = 2 \;\; T_p^2 = 2 \;\; T_p^3 = 3$$

then we must count 2 twice:

$$S = \frac{10}{\sqrt[3]{2 \times 2 \times 3}} \approx 4.37$$

Other applications

The definition is useful for randomised search algorithms, where T_s may also vary considerably.

It also covers the case where we wish to calculate the speedup of one parallel program $\mathcal{A}$ over another parallel program $\mathcal{B}$, if we take T_s^i to mean the parallel execution times of $\mathcal{B}$.

It even covers the case where programs $\mathcal{A}$ and $\mathcal{B}$ take different queries. The times T_s^i and T_p^j may be the (parallel or sequential) times for a set of queries, in which case S is an average of the speedup of $\mathcal{A}$ over $\mathcal{B}$ for those queries.

Quasi standard deviation

To measure the deviation from the geometric mean Ertel suggests the *quasi standard deviation*:

$$D = e^x$$

where

$$x = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m} \frac{1}{mn} \left(ln \frac{T_s^i}{T_p^j} \right)^2 \;-\; \left(\sum_{i=1}^{n}\sum_{j=1}^{m} \frac{1}{mn} ln \frac{T_s^i}{T_p^j} \right)^2}$$

Again if some of the times are the same, count them as if they are different.

If $D = 1$ then there is no deviation from the mean, in contrast to the usual standard deviation which is 0 when there is no deviation.

5.3 Speedup curves

For performance analysis we often plot a graph of speedup as a function of the number of workers. However, there are a few pitfalls which should be avoided:

- As mentioned above, speedup may vary from run to run, and so to plot a speedup curve we should several runs for each number of workers and find average speedups.
- Speedup curves may take any shape; for example there may be kinks, troughs, or plateaus. This means that speedup curves cannot be extrapolated nor interpolated. For example, if we have a nice, linear speedup curve for $1 \ldots 10$ workers we cannot use this as evidence for a good speedup with 11 workers.
- Even a small change in a program *may* have a large (good or bad) effect on speedup. Therefore we may get a completely different speedup curve after a small change, (in the program, the query or the number of workers for example.

These effects (which are described more fully in an earlier technical report [Pre93b]) show that a speedup curve actually says little about a parallel execution. A good speedup curve does not necessarily indicate good parallel behaviour, and so caution should be exercised when using speedup curves. This is not to say that speedup curves are useless: we can take a poor speedup curve to indicate poor parallel behaviour.

Acknowledgement

Thanks to Liang-Liang Li, Micha Meier, Shyam Mudambi and Joachim Schimpf for suggestions and proof-reading.

Chapter 3.

PSAP : Planning System for Aircraft Production

Jacques Bellone

1 Introduction

PSAP (**P**lanning **S**ystem for **A**ircraft **P**roduction) is an *aircraft production planning decision support system*, whose aim is to schedule the aircraft production over the next years (more than five years). Up to 400 aircraft and 30 assembly lines are concerned. Among the several factories involved, the Argenteuil factory was chosen as a pilot site, since, in the current practice, the production plans for the other factories are generally derived from those elaborated for Argenteuil. The users are the planning experts of the Argenteuil factory.

The planning process takes into account all factory departments involved in the main assembly steps, whose results will condition all further production management computations (schedule for each factory, required primary parts, workshop schedule computations...).

Such long term production schedules the assembly lines where the big aircraft sections are manufactured. A section is a major aircraft part (e.g. cockpit, wing, rear fuselage, final assembly). This production is paced.

The objective is to find schedules respecting the delivery dates for all aircraft and being a satisfactory compromise between section storage costs and workload. The production plans have to take into account the assembly lines' limited flexibility (not all production rate transitions are possible and each significant change has a considerable cost).

Production planning and scheduling are complex operations involving a great number of constraints, both numerical and symbolic. These constraints are likely to vary with time: constant changes in the production context make it necessary to modify the constraints embodied in the planning system, as well as the cost elements. In such environments, classical Operational Research programs and ad hoc software definitely lack flexibility. In contrast, Constraint Logic Programming [Col87, JL87, DSV87] appears as a good candidate, since it offers the desired modifiability as well as efficiency.

1.1 Production Intervals and Assembly Lines - Definitions

The *production rate* of an assembly line is the number of aircraft sections produced per month.

The *production interval (PI)* is the number of days between the start of two successive aircraft sections on an assembly line. In other words, each assembly line receives a new aircraft section every *PI*. *PI* is the reciprocal of the production rate.

An assembly line is dedicated to the production of a particular section of one or more aircraft types (e.g. single seater aircraft, twin seater).

1.2 Current Practice

Preparing a production schedule is currently performed by a planner with a pencil and an eraser. It takes one week, which raises essentially two problems :

1. It is a very long and inflexible process : The planning task occurs, basically, every three months, but there is a constant requirement on planners to *simulate* the effect of possible new orders, of confirmation or cancellation of options, of modification of due dates or of the main characteristics of certain aircraft, etc... They can usually only provide approximate answers by testing limited local modifications to the current schedule. Furthermore, such a job takes from 1/2 day to 3 days, which is rather dissuasive.

2. The result can only be evaluated a posteriori : As the cost criteria cannot be formalized, the planning process can only take into account intuitive cost criterion. This means that the result cannot be characterized precisely with regard to these criteria.

Thus has emerged a need for a planning tool that would not only compute schedules according to a given formalized evaluation function, but also be flexible and swift enough to be used as a decision support system for evaluating any interesting scenario upon request. The scenarii might be for current production (e.g. for the Mirage 2000), as well as for pre-production (e.g. for the Rafale). For pre-production, delivery dates and some production parameters (e.g. production cycle parameters) are often changed to test several production possibilities.

1.3 PSAP History

When the Dassault factories started to look into the question what kind of software tool would help the production planner, their first attempt was to adapt the ARTEMIS [Sys88] software to the planner's need. But it soon appeared that this main frame scheduling software would not be able to really solve their problems (cf. chapter 3).

This motivated internal development of a decision support system for planning with the objective of automating certain tasks and of anticipating the evolutions of the production context. This development was assigned to the *Artificial Intelligence and Advanced Computer Techniques Department* of the *Advanced Studied Division*. The emerging Constraint Logic Programming, which seems well suited to handle planning and scheduling problem, was chosen (for the reason cf. chapter 3).

Starting from very few specifications, a first mock-up, called PSAP 1 (cf. chapter 5), was developed. The modelling of the problem was naive. The results provided by this mock-up were sufficiently good to demonstrate the feasibility of a CLP based system for this type of complex problem.

A first prototype, PSAP 2 (cf. chapter 5 and 6), based on precise specifications was then completed. Parallelism was applied to this prototype to search some optimal costs. The results were not satisfactory neither in terms of quality of the solutions nor in terms of parallelism use. The search tree was too large, even for parallel execution, the properties of the problem were not sufficiently analyzed.

The PSAP 3 prototype was developed (cf. chapter 6) in order to reduce the search tree. The pre-computation and the labelling phases were modified. The parallelism was applied to PSAP 3 and gave interesting results.

1.4 Summary

Chapter 2 describes the application in terms of objectives to be fulfilled, inputs/outputs and the planner's work.

Chapter 3 discusses the limits of an "on the shelf" scheduling tool such as ARTEMIS, the choice of Constraint Logic Programming, the size of the problem and its complexity, first solution vs. optimal solution, why parallelism was necessary.

Chapter 4 studies the first constraint modelling of the problem(PSAP 1).

Chapter 5 studies the parallelism introduced to solve the cost optimization problem : where it has been introduced, introduction difficulties, methodology used and parallelization effort.

Chapter 6 studies different pre-computations and heuristics through PSAP 2 and PSAP 3. Sequential and parallel improvements to PSAP are discussed, followed by a description of their interaction. The improvement from PSAP 2 to PSAP 3 was driven by the performance debugging process, as shown in the benchmark paragraph.

2 Problem Description

2.1 Objectives

The objective of PSAP is to provide a decision-making support system for Dassault's production planning experts (hereafter called planner).

The first task of the planner is to pace the different assembly lines assigned to the manufacture of given aircraft units in accordance with their due dates, with a view to preparing the production schedule. Then, the planner has to calculate the resulting workload and storage time.

The overall goals are :

- balancing the workload,
- reducing staff changes (i.e. the number of production interval changes),
- reducing storage costs.

Balancing factory workload is quite impossible within PSAP. As a matter of fact, PSAP's aim is to pace the assembly lines for one given aircraft model (Falcon, Mirage 2000 or Rafale), whereas Dassault factories produce all these models.

Staff changes increase the workload but reduce the storage costs.

As one may imagine, it is not easy to achieve a compromise between these three goals. Moreover, there is no cost function which takes into account both storage costs and staff changes. The difficulty in formalizing a cost function leads to the design of an interactive system. These requirements imply for the system to be **easily parameterizable** and

to interact with the user via a user-friendly interface. User interactions require that schedule solutions must be obtained within a short time.

2.2 Inputs

Three kind of inputs are used :

- Aircraft Production rules, also called Factory Data. The most relevant factory data are description of the assembly lines, description of the different aircraft production types, and for a given type of production, description of the related sections (production cycle, workload, precedence constraints, storage costs). A section is a major aircraft part (e.g. cockpit, wing, rear fuselage, final assembly) produced in one assembly line.
- Production Orders, also called Planning Data. They are objective data. The most relevant planning data are delivery due dates for the orders, work in progress on the assembly lines, learning curve laws (i.e. a new production or a staff change implies an increase of the production cycle during the time the staff is learning new tasks).
- Production Policy, also called Planner Data. They are subjective data. The most relevant planner data are the requirements stated by the factory director in terms of workload and/or of storage costs, the planner's expertise, his wish to avoid too frequent production interval changes as well as changes to the short term production. These subjective data need an interactive system and the introduction of subjective parameters to be set by the planner.

The planner data are not predictable and require interactions during the scheduling process, whereas the factory and planning data are described a priori.

2.3 Outputs

The main output is the schedule for each assembly line. This is the objective of PSAP. But the overall goal is to have a schedule which gives a good compromise between the staff changes and the storage time for a each aircraft model with a balanced workload for each factory. The definition of the good compromise depends on the current production policy.

Hence, the other main outputs are :

- the number of the production interval changes,
- the storage time for each section, each aircraft and each assembly line,
- the workload and the overload for each section, each aircraft and each assembly line.

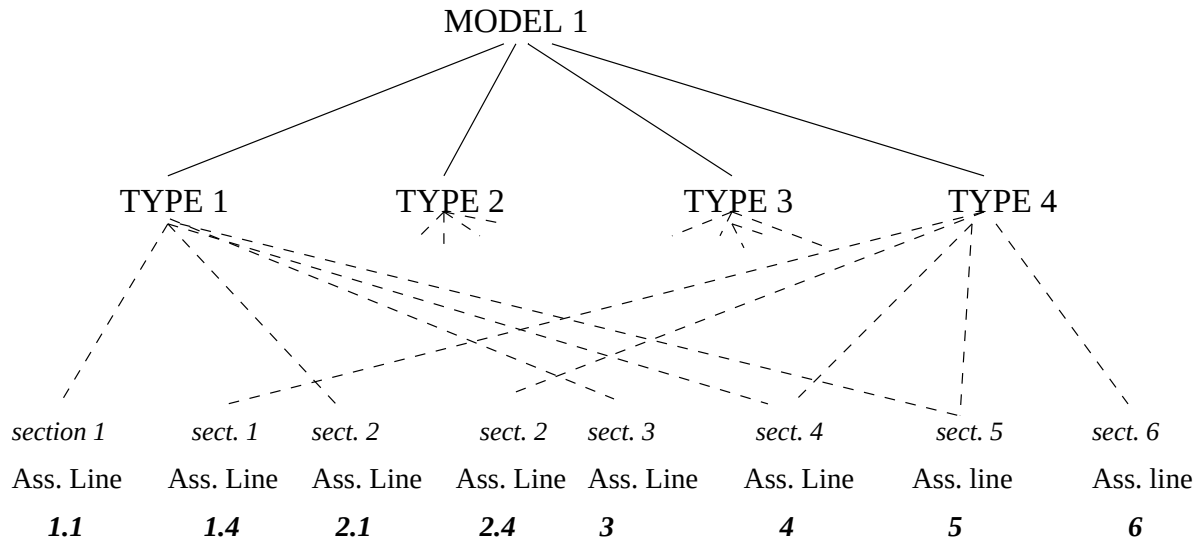


Figure 2.1: Example of Factory Data, one aircraft model production type has 5 sections produced by 5 assembly lines (sections and assembly lines are production type-dependent)

2.4 Cost Functions

The assessment of a schedule depends on the results given by the different cost functions.

A schedule involves two costs : **workload** and **storage time**. These costs qualify the schedule's qualitative properties. However, such qualification is actually not an easy task, as balancing these two costs in one and the same cost function is not easy to achieve. Moreover, balancing these two costs depends on the current production policy.

The workload of an assembly line depends on its different production intervals and on the workload for each of its manufactured sections. Moreover, when the production interval changes, there is an overload due to the workers' new task learning time.

The workload used for a given section depends on :

- the rank of this section on its assembly line (this rank is the parameter of the staff learning function which is a logarithmic function),
- the last production interval change which produced an overload.

The overload function for a given section, which is a further learning function due to the staff change, depends on :

- the time interval between this section and the section where the last production interval change occurred; if this time interval is shorter than the latter's production cycle, an overload occurs for the section,
- the ratio between the current production interval and the previous one.

The complexity of the workload cost function for an assembly line has led to restrict the workload optimization to overload optimization in PSAP and hence, to minimize the production interval changes.

The storage time is the time between the end of manufacturing one section and the manufacturing start of the next section.

The relevant costs actually used in PSAP are **the number of production interval changes** and the **storage times**.

2.5 Details of the Planner's Work

First task, the planner schedules a new production or modification of the aircraft to be produced (modification to some aircraft type, adding or removing aircraft). In this case he schedules the last assembly line, then the previous one and so on. This allow minimization of the storage time between the end of an aircraft production and its delivery date. To start by the last assembly line is not a simplification, it is the actual way of scheduling. This way of scheduling is the most efficient because in long term scheduling, the last assembly line is the most constrained.

The subtask, which schedules one assembly line, consists in pacing each assembly line, i.e. in choosing for which aircraft section the production interval has to change and which value the new production interval has to take.

Second task, he calculates the resulting workload and storage time.

Third task, considering a previous schedule, he may want to modify some of the assembly lines which may imply modification of the next and/or previous assembly line(s). This modification is mainly due to the fact the work in progress in this assembly line has not respected the previous schedule. The aim of the planner is to minimize the number of assembly lines to be rescheduled.

The second task may occur at any time during task 1 and 3.

Forth task, often included in the first or the third task, the planner optimizes schedule costs with regards to the current production policy.

The subtask "to pace an assembly line" might be more or less difficult to calculate. Scheduling the last assembly line, called general assembly, requires a lot of work. When the assembly line scheduling is done from the last line to the first one, a given assembly line (which is not the last one) is easier to schedule if :

- the same aircraft are produced by this line and the following line,
- there are no learning curve laws applied to the production cycle of sections produced by the following line.

But as can be seen in Figure 1, the first condition is respected by 60-70% of the assembly lines (e.g. not respected by lines 2.4 and 3 which share aircraft from line 4). And the second condition is respected only by old production without any new production type or new aircraft version.

Thus, at least 50% of the assembly lines need a lot of scheduling work, and for a brand new aircraft, this could be 100% of the assembly lines.

3 Qualification

This chapter discusses :

- the limits of classic scheduling tools such as ARTEMIS [Sys88], a project management product which handles scheduling, planning and resources management aspects,
- the choice of CLP,
- the problem size and complexity of the problem,
- first solution vs. optimal solution,
- limits and optimization, which led us to consider parallel CLP.

3.1 Limits of ARTEMIS

ARTEMIS has been studied by the Argenteuil and Merignac factories' planners.

The main drawbacks noted at that moment were :

- no possibilities to optimize cost functions,
- no possibilities for easy handling of the production interval,
- ARTEMIS has been adapted to be able to manage the problem in the same way as without software.

While Merignac's planners chose ARTEMIS as a drawing tool, Argenteuil's planners preferred to search for a more powerful software package. Now Merignac's planners also want PSAP.

3.2 Why CLP ?

Constraint Logic Programming (CLP) expresses in a very declarative and comprehensive way the constraints and successfully manages combinatorial problems encountered in production scheduling (e.g. the car-sequencing problem).

CLP thus preserves the declarativity of PROLOG, allowing swift software development and offering easy software modifiability, but greatly improves the resolution speed on highly combinatorial problems by hard-wiring of domains and constraint propagation.

It allows problems to be addressed in a flexible way, which so far have only been solved by rigid conventional programming methods (as Operations Research methods). Among these problems are production management, planning and scheduling, logistics.

Thus, the first PSAP mock-up has been implemented using CLP. The following paragraphs summarize the experiments with this sequential version of PSAP.

3.3 Problem Size and Complexity

The average size of the problem manageable by the PSAP prototype is to plan 200 planes and 12 assembly lines over five to ten years. The operational system needs to handle up to 400 aircraft and 30 assembly lines over five to ten years. But, as the schedule is computed assembly line per assembly line, the maximum size of the PSAP problem is to pace 400 aircraft sections over 10 years with a time unit of a half day.

A PSAP complexity analysis compared PSAP to the Warehouse Location Problem. This study took into account the schedule problem as a whole (i.e. with the workload and overload functions). The conclusion, even if there is no formal proof, was "it seems that PSAP is at least as difficult as a Warehouse Location Problem with variable size" [BS93]. The whole PSAP problem hence seems NP-complete.

Because of this foreseeable difficulty, the whole problem has never been taken into account by the PSAP prototype. Due to the impossibility of balancing the two costs, PSAP prototypes solve a simplified problem using several steps. Thus, the planner shares his work into several tasks : pace one assembly line, then look at its storage and workload costs.

Considering this simplified problem, a polynomial algorithm might be found. But its execution time would be rather dissuasive due to the combinatorics introduced by the number of solution possibilities as briefly explained above.

The combinatorics is due, when pacing an assembly line, to the choice between increasing, decreasing the production interval, and keeping it even.

If these 3 choices are available for each aircraft section, they imply, for 250 aircraft units, a huge 3^{250} search tree. A polynomial algorithm must obviously prune this tree but this will not be sufficient for the actual data.

But such an algorithm will need a cost function which the planners are not able to fix.

3.4 First Solution vs. Optimal Solution

A schedule involves two costs : **number of interval changes** and **storage**. As the relative weights of these costs are likely to vary with the context, PSAP can optimize either storage cost or the number of interval changes, or number of interval changes then storage cost; it is up to the planner considering his current policy. He can also interact with PSAP to optimize these costs from a production policy point of view using his knowledge.

He can mix these interactions with the provided automatic cost optimization. The subjective PSAP parameters are used for both manual and automatic optimization.

The subjective PSAP parameters have to be significant for the planner and must provide a "good first solution" in terms of costs. A good first solution minimizes both the number of interactions during the manual optimization and the automatic optimization time.

For instance, the most appreciated of these subjective parameters allows the planners to give the maximum number of interval changes before pacing an assembly line.

Thus, pre-computation and heuristics manageable by the planner with understandable parameters is a way to add constraints in order to provide acceptable-quality first solutions.

3.5 Why Parallel CLP ?

In the sequential CLP version of PSAP, the first experiments have shown two drawbacks :

- the optimization time is too long even if optimization started with good bounds,
- the quality of the schedule is too sensitive to the input data.

This is true for the first mock-up and also for the PSAP 2 prototype with the first attempt to introduce some pre-computation and heuristics. The constraint modelling part will detail why, but at this point we have felt it difficult to sufficiently prune the search space to allow efficient time optimization with sequential CLP.

The remaining search tree is still sufficiently large to feel that there is a need for parallelism when searching an optimum, for the following reasons :

- the aim of pre-computation and heuristics is to prune the search tree (and thus, to decrease the number of solutions) without removing interesting solutions, as it is difficult to know a priori where these solutions are, the search tree is not sufficiently pruned to be explored in sequential execution,
- the pre-computation implies non-deterministic research the size of which depends on the pre-computation (subjective) parameter value (value set by the planner).

The need for parallelism has also been felt necessary as optimization proved possible with small-size data sets (e.g. 70 aircraft), however not for large-size data sets (e.g. 250 aircraft).

4 Constraints Modelling / Initial Prototype

The initial mock-up (PSAP 1) was followed by two prototypes (PSAP 2 and 3) which have explored several pre-computation, heuristics and parallelism possibilities.

The PSAP kernel is divided into 3 main parts :

- precedence constraints between the different sections of one and the same aircraft,

- production interval constraints between the sections produced in an assembly line,
- procedure to label the start date of each section in an assembly line.

The first part of PSAP is trivial for CLP techniques.

The second part of PSAP is more complex due to the disjunctive constraints expressing the choice between increasing, decreasing the production interval and keeping it even.

The precedence and interval constraints obviously prune the huge search tree but this will not be sufficient for some of the data. This search tree is explored within the third part.

The third part of PSAP chooses where and how the interval is changed. This is called the labelling part, and the search tree is explored in this part. Hence, parallelism is introduced in this part. Cost optimization is implemented in this part as well.

4.1 PSAP Constraints

Precedence Constraints

These are precedence constraints between macro tasks (airframe sections) for each aircraft.

The sequencing of the assembly is partially ordered. Below is shown a schematic precedence flow chart for the Mirage 2000. "$A \rightarrow B$" means "A is before B", and this is a constraint. Note also that the assembly, e.g., of structure subsections "1-17" and "17-26" makes section "1-26".

These constraints relate to two types of constraints :

- Cycle length constraints, expressing a task duration ($ProductionCycleSection_i$) within which it is impossible to begin the next one ($BeginDateSection_{i+1}$).
- Delivery date constraints, imposing a maximum length to the whole production cycle.

These constraints between a section i and the following section $i+1$, look as follows :

$$BeginDateSection_i + ProductionCycleSection_i + StorageTimeSection_i$$

$$\# = BeginDateSection_{i+1}$$

In the case of *delivery date constraints* $BeginDateSection_{i+1}$ is the delivery date of the aircraft, and i is the last section of it.

Production Interval Constraints

These are constraints between the planes on each assembly line.

The number and the extent of changes to the interval are strictly limited once the standard production speed has been reached. Precise figures will be given, e.g. at least 3 months

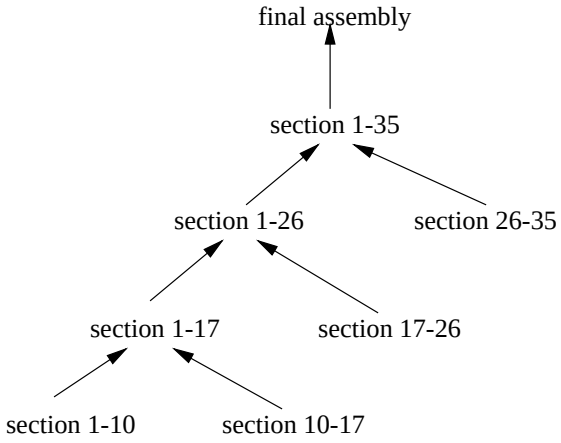


Figure 4.1: Precedence Flow Chart

between two consecutive changes; do not decrease the interval by more than $X\%$ for initial value between A and B, and so on ... These interval-changing constraints will be called **line-interval constraints**.

There are additional constraints, complying with the capacity limits, over the planning horizon, of the assembly lines. They impose, due to the unicity of the tools, a minimum interval of time between aircraft of a certain type on one assembly line. These constraints shall be called **type-interval constraints**.

All these constraints are disjunctive constraints.

4.2 Domain Size

PSAP sets the precedence constraints between the sections of each aircraft. This is straightforward and allows the domain-size available for the start date of each section to be reduced to the maximum value of the storage time (e.g. 120 half days, data set by the planner). Then it computes the assembly line planning per assembly line, beginning by the last one. The constraints on the production interval reduce the domain for the start date (e.g. 90 half days, if started at 120). The size of the actual problem to be solved for each assembly line thus is to find a value for the start date of N aircraft sections in a 90 domain-size. N is the actual number of aircraft to be planned and currently is between 100 and 300.

4.3 Limits of the First CLP Implementation

The planners' fuzzy specifications for the schedule costs had led to a very disappointing mock-up :

- the quality of the schedule was far from the one expected by the planner, (mostly because there were too many production interval changes),

- the time constraints did not sufficiently constrain the system, which resulted in a lot of uninteresting solutions,
- the costs constraints were too weak, hence no actual optimization (neither storage time nor workload) was possible, with the naive modelization chosen, in an acceptable time (this long time was also due to the huge number of solutions).

The two first points led to an uninteresting first solution, the last one shows that a sequential optimization is unacceptable by the planner in terms of time.

Moreover, the lack of definition of the cost criteria requires a parameterizable and interactive tool capable providing an acceptable first solution. Only then a tool with optimal solution search can be considered.

4.4 Conclusion

PSAP 1 was a sequential mock-up version ported from CHIP to ElipSys. No heuristics were implemented. Poor cost specification and too big a search space led to a great amount of useless backtracking at the end of the schedule. This huge search space implies that there is limited optimality search time. The cost found was thus in fact a *suboptimal cost* (cf. paragraph 5.3) far from the actual optimal value.

The main problem of PSAP arises from the lack of time (i.e. *precedence* and *interval*) constraints and of costs constraints. Parallelization, pre-computation and heuristics have been used to reduce this combinatorics.

Considering PSAP 3, its use of parallelism, pre-computation and heuristics, the expressive power of ElipSys or ECL^{i}PS^{e}is adequate.

5 Parallelization

Parallelism was introduced into the first sequential prototype in order to solve the cost optimization problem. The sequential prototype provides suboptimal solutions only for small-size data sets. These solutions were moreover far from the solution expected by the planner.

The following paragraphs discuss where parallelism was introduced, parallelism introduction difficulties, the methodology followed to introduce parallelism and the last paragraph concludes on the parallelization effort.

5.1 Where is parallelism introduced ?

As searching for the first solution is fast, parallelism is only needed when an optimal solution is searched with, for instance, a branch and bound procedure as the one given by the *min_max* built-in.

Two parts seems well suited to support parallelism so as to improve the PSAP execution time and results : the *interval constraints* part and the *labelling* part.

In the *interval constraints* part, setting the disjunctive constraints in parallel might avoid the drawback arising from the use of the disjunctive constraints. But the size of the tree would remain the same (3^N , with N= number of aircraft) and only a huge number of processors would improve the execution time. This possibility was hence not studied further.

As the search tree is explored in the *labelling part*, some parallelism may improve this part. But parallelism alone will have the same drawback than in the *interval constraints* part. The solution is to mix parallelism with more or less lazy heuristics. PSAP 2 started this work with lazy pre-computation and heuristics which still needed too much parallelism. Then, a better compromise was studied in PSAP 3 with eager pre-computation and heuristics which need less parallelism.

This identification and coding is quite easy; the main effort is **to predict the improvement due to parallelization** and **to tune parallelization and heuristics**.

5.2 Parallelism Introduction Difficulties

"Ideally, the effort required for parallelizing a sequential program should be limited **to identifying** those portions of the program that must be executed in a parallel fashion and **making sure** that they actually run in parallel without communication overhead" [Mud94].

The **identification** phase is quite easy, at first sight, and ECRC has provided us with some documents about this phase [Pre92].

The **checking** phase needs a tool such as *ParSee*. *ParSee* provides visualization of the parallel execution behaviour to check the amount of used (or needed) parallelism and the size of communication overhead.

For PSAP the *checking* phase was not so easy. Even if benchmarks have shown no communication overhead and further potential for more processors than allowed by the used computers, the access time to an optimal (or suboptimal) solution was too long. This was mostly due to the fact that parallelism has been introduced to solve a point that a sequential program does not really manage : the search for the optimum. The problem, in its first modelization, was so hard that even parallel execution with 4-processor or 12-processor computers did not fast enough the optimality search.

In fact, the introduction of parallelism, starting from a first prototype, implies two different questions :

- is the size of parallel grain coarse enough ?
- is the size of the remaining search tree (pruned by the constraints) small enough ?

Answering the first question leads to making choices on the nodes in which parallelism will be added, and in which way.

Answering the second question leads to working on the constraints modelization, on the labelling heuristics, on the branch and bound strategy and, if needed, to adding a pre-computation step. This work has, of course, effects on the above choices; this is why, in the following paragraph on methodology, the first step concerns sequential behaviour.

5.3 Methodology

When parallelism is introduced to solve a cost optimization problem not tractable under sequential execution, two points have to be solved :

- how parallelism has to be introduced,
- which maximum data size can be handled by the parallel program.

Our benchmarking work followed two methodology directions with strong interactions.

The first direction used aims at tuning the program behaviour and to introduce parallelism in the best conditions. This methodology may be called **parallelism introduction methodology** and has three steps :

- first step is to obtain good sequential behaviour : constraints modelization, pre-computed constraints, labelling heuristics, branch and bound strategy, in order to prune the search tree;
- second step is to choose the nodes where parallelism will most improve the program;
- third step is to choose the actually needed number of *workers* (processors).

The first step is described in the *CHIC Lessons on CLP Methodology* [CFGG95b].

The second step is described in the APPLAUSE report [Pre92].

The third step needs more inputs. Many benchmarks on the whole range of data sets are needed for making this choice. For each benchmark, a *ParSee* analysis has to be done to answer this question.

The second direction used aims at fixing the maximum data size the parallel program can handled. This methodology can be called **benchmarking methodology**. It will give the limits of the parallel program in terms of specification (i.e. are current data sets handled ?) and in terms of qualification (i.e. what are the improvements provided by parallelism, pre-computation, heuristics... ?).

Each of the *parallelism introduction methodology* steps needs benchmarks on the relevant data sets. *Benchmarking methodology* mostly consists increasing the size of the search tree. This size is increased according to two directions :

- data size : start with small data sets and increase the data set size,

- search tree size : for each data set, start working without (sub)optimal search, then go on with suboptimal search; lastly, if the search tree is sufficiently pruned, finish with optimal search.

There are different kinds of suboptimal search :

- search in limited time,
- search skipping over solution less well than a given percentage of the found suboptimal cost,
- search where the minimum bound of the search space is too high.

In the two first kinds, the suboptimal search is known a priori.

In the last kind, we do not know a priori what kind of search will be done. In this kind, the difference between a suboptimal search and an optimal search depends on the value of the found solution : if this value is lower than the minimum bound it is a suboptimal search, if not, it is an optimal search.

Moreover, this last kind can be found into the two first cases.

The different ways to limit the search tree size hence are :

- to start with a low maximum bound (the first solution found in a previous trial) and to increase it,
- to start with a high minimum bound and to decrease it,
- to prune the search tree from a given percentage below a found suboptimal solution,
- to stop the search after a certain amount of time. The latter way may also be a requirement of the system in order to give constant answer time to the end-user. The only drawback is how to make sure that a solution will be given within this time. This feature is now available in ECL^{i}PS^e.

5.4 Conclusions on Parallelism Introduction

"Ideally the effort required for parallelizing a sequential program should be limited to identifying those portions of the program that must be executed in a parallel fashion and making sure that they actually run in parallel without communication overhead."[Mud94]

The current technology would be very close to this ideal if the following points were more straightforward :

- the forecast of the biggest OR-nodes,
- the forecast of the parallel behaviour of a sequential program.

If the introduction of parallelism has not significantly added time as needed to debug the system. However, it has increased the time to understand the behaviour and the results of the program and its reliability. This performance debugging requires significant time.

This significant time is due to the difference between a suboptimal search and an optimal search described below. For instance, in PSAP 2 which prunes the search tree from a given percentage below a found suboptimal solution, time was spent :

- on finding a percentage for each data set,
- on finding benchmarking where the successive suboptimal solutions are identical whatever the number of *workers*, otherwise speed-ups between executions which give different results are not meaningful,
- on finding the minimum lower bound.

From a developer's point of view, parallelism has allowed the optimization problem modelization to be studied (which would not have been feasible in sequential execution) and thus, has improved the sequential version of PSAP.

6 Performance Debugging / Improvement

It is difficult to split sequential and parallel improvements. Both are linked by a common program behaviour, i.e. the sequential modelizations can provide different biggest OR-nodes (different by their size or/and their place in the search tree).

Nevertheless, as the basis of a parallel program is a sequential program, this chapter describes sequential improvements, and then parallel improvements in its two first paragraph. The third paragraph discusses the result of these different improvements and their interaction through a representative set of benchmarks.

All conclusions about sequential or parallel improvements are the results of several benchmarks on PSAP 2 and PSAP 3. But the benchmarks sections only show the main results with parallel execution and discuss the obtained speed-ups.

Real data were used at each development step. Only the size of the data was reduced (70 aircraft instead of 200) in order to speed up performance debugging (and to have results when optimizations are too long with PSAP 1 or PSAP 2).

6.1 Sequential Improvements

As PSAP 1 constraints do not propagate enough to give results which satisfy the planner, a pre-computation has been added. The aim of this pre-computation is to add some constraints which prune the search tree.

Two stages succeed one another :

- the first pre-computation, implemented for PSAP 2, is a lazy pre-computation which leaves large search tree space; Parallelism is expected to improve this labelling procedure when searching for a (sub)optimal cost;
- the second one, implemented for PSAP 3, is an eager pre-computation with a much smaller search tree than the PSAP 2 pre-computation. In this case, parallelism is needed to improve the reliability of the pre-computation results when searching for a (sub)optimal cost.

PSAP 2 Pre-computation

In **PSAP 2**, the first prototype version tested in the factory, more user control together with pre-computation, heuristics and more parallelism, were introduced. With a given maximum number of production interval changes determined by the user, a pre-selection of possible change ranks is achieved, with some rating used as a starting point for the heuristics.

To obtain more pruning, a statistical pre-computation is made in order to simulate the reasoning so that it can be identified by the planner's eye. This pre-computation is made once the interval constraint has been set, giving two types of information indicating :

- the aircraft units, not requiring a production interval change, to be eliminated,
- the possible production interval for the remaining units.

The heuristics using the information from this pre-computation are lazy heuristics, in the sense that the search tree is pruned but the final decision is still taken by the constraint propagation during the labelling part. Such heuristics were chosen because :

- at that moment, an optimistic view of parallelism with constraints led us to think that constraints and heuristics would provide some good decisions to prune the search tree and that parallelism would sufficiently speed up the search through it,
- the eliminated aircraft are actually to be eliminated, i.e. no "good" solution is removed by this pruning.

Pre-computation results, as arising from statistics, prune the search tree by adding new constraint propagation.

Stating constraints to forbid interval changes on aircraft not found by the statistical pre-computation really prunes our search tree, for instance, with N = number of aircraft, from 3^N to a $3^{N/10}$ (e.g. 3^{250} to a 3^{30}) branches without removing any relevant branches.

Pre-computation results are also used to improve the labelling procedure.

PSAP 2 Labelling

PSAP 2 labelling expresses the same three possibilities as PSAP 1 labelling (to increase, to decrease, to keep even the production interval) but sets the production interval to the value given by the pre-computation as first value.

The use of the pre-computation production interval by the labelling procedure does not really prune the tree. However, at least in sequential mode, this use avoids a lot of backtracking as, most often, the first production interval chosen within this labelling is an acceptable solution whereas the one chosen by other domain value labelling procedures is not an acceptable solution.

The other domain value labelling procedures tested are the three following ones :

- maximum domain value domain taken as first production interval; this labelling minimizes the storage cost but increases the number of interval changes,
- minimum domain value domain taken as first production interval; this labelling minimizes the number of interval changes but increases the storage cost,
- central domain value domain taken as first production interval; this labelling tries to find a compromise between the two costs.

The choice of the pre-computed value instead of the central value as first value when labelling the production interval, has provided PSAP 2 with best compromises between the two costs. This labelling tries to put the suboptimal solutions in the left part of the search tree.

This labelling is hereafter called **interval direction**.

Sequential PSAP 2 Conclusions

This pre-computation is tuned by a subjective parameter actually hard to use by the planner. The effect of this parameter is to forbid a interval change to more or fewer aircraft but the number of the remaining aircraft are out of control for the end user.

As a conclusion, a value had been fixed for this parameter being the one used while benchmarking.

Lack of precision when tuning the parameter leads to dramatic behaviour.

For instance, the same value of this parameter applied to some data leaves too many remaining aircraft (e.g. 30); in this case, the search tree is not sufficiently pruned; whereas applied to some other data, it does not leave enough aircraft (e.g. 4), in the latter case, some "good" solutions disappeared from the search tree.

Moreover, it appears in both cases that this parameter can, sometime, remove "good" solution branches. This is due to the difficulty for a statistical function to be reliable in all cases.

Due to the lazy pre-computation, *interval direction* labelling needs parallelism to find suboptimal solutions in a search tree which is not sufficiently pruned for a sequential execution.

But this lazy pre-computation prevents the search for the optimal solution of actual data sets because the search tree is still too large.

Pre-computation (*slope_difference*) and Labelling in PSAP 3

As the planner requested a better first solution, the statistical pre-computation was modified in such a way that the new pre-computation (called **slope_difference** because difference of latest start date slopes is the basis of this computation) only keeps a few aircraft. Then the heuristics during the labelling will be "eager" and no longer lazy, as in PSAP 2.

Pre-computation acts in two steps :

- the first step sets a weight on each aircraft, the *heaviest aircraft* being the most interesting interval change points,
- the second step, driven by the end user, filters the *heaviest aircraft*.

In other words, if the end user requested 6 changes, the second step of the pre-computation will filter the 6 aircraft where interval changes are required, and the *interval direction* labelling is optimized to achieve a storage time that is as small as possible. Then, using the PSAP interface, the end user can change the schedule as he wishes.

Moreover, as the pre-computation is safer and as the production interval is dynamically computed at the beginning of each labelling, the labelling sets the production interval value which obviously decides whether the interval is to increase, decrease or to be kept even.

As the *interval direction* labelling is optimized, there is no need to parallelize it as in PSAP 2. The introduction of parallelism will arise in the new cost optimization procedure.

Even if this pre-computation is much more reliable than that of PSAP 2, there is no proof that the *heaviest aircraft* are really the best points for production interval changes. The new cost optimization procedure takes into account this uncertainty and removes it.

6.2 Parallel Improvements

This paragraph discusses where the parallelism was introduced in PSAP 2 and PSAP 3.

The lazy pre-computation of PSAP 2 implies a huge search tree during the labelling phase, hence parallelism was introduced into the labelling procedure in several ways.

The eager pre-computation of PSAP 3 and its new labelling implies a much smaller search tree in the labelling phase, hence parallelism is not needed in this phase. Parallelism is needed in a phase just before labelling. The aim of this phase is to remove the pre-computation unreliability.

Introducing Parallelism and Heuristics in PSAP 2

The labelling part was parallelized, thereby allowing searches in the 3 branches at the same time. This procedure thus expresses the three interval labelling possibilities :

- to increase the interval,
- to decrease the interval or
- to keep the interval even;

and sets a value to the production interval.

Parallelism may, in addition, overcome the pre-computation result drawback due to the statistic, not always reliable, solution.

Several labelling procedure were tested :

- the best sequential heuristics (called *interval direction*)
- mixed parallelism, to express the three interval labelling possibilities, and heuristics, to set the production interval value (called *parallel interval direction*),
- all in parallel (called *parallel_3*),
- only production interval value setting done in parallel (called *parallel 1*),
- more or less parallelism, called e.g. *parallel_3_100* : parallelism is used only for the 100 first aircraft.

The aim was to know how to tune parallelism in such a CLP application, i.e. how much parallelism is needed to actually improve the results.

The used heuristics parallelize either the choice of the new production interval (e.g. parallel_3 and parallel 1), or not (e.g. *interval direction* and *parallel interval direction*); and they parallelize either the choice of the production interval changes (e.g. *parallel_3* and *parallel interval direction*) or not (e.g. *interval direction* and *parallel_1*).

But no heuristics can completely determine where and how to change the production interval and the remaining disjunctive constraints do not allow good constraint propagation, i.e. the production interval domain-size reduction is done only during the labelling (overall by the way of an element constraint).

PSAP 2 Conclusions

Whatever the parallelism and heuristics used, the search tree remained too large and the access time to a suboptimal solution too high. Even if *super linear* speed-ups, i.e. for N *workers* the speed-ups are greater than N, were noticed and if *ParSee* analysis shows that better results could be reached with a 30-processor machine, the access time to a solution and the unreliability of the pre-computation led to the introduction of the PSAP 3 pre-computation which requires to parallelize another predicate.

New Cost Optimization

The use of the *slope_difference* pre-computation results in two ways being available for use of the *min_max* primitive. The first way really improves the storage cost starting from a mean cost, the second slightly improves the cost found by the first one. These two ways have to be used in the following order (with P the number of possible values for a production interval) :

- instead of choosing 6 aircraft out of 30, the end user may, in a first stage, report 10 changes and then, request the optimal solution in terms of storage days with only 6 changes out of the previous 10 (the search tree is then reduced to $210 * P^6$ branches instead of 10^{11})[2].
- to improve a result in terms of storage, the planner sometimes just moves the change point to 1-3 aircraft before/after the first change point found. The search tree size for 6 changes, if we allow 5 possible aircraft for each change, is 5^6 ($15,625 * P^6$ branches).

In fact, the first point allows to get rid of small errors, i.e. the uncertainty of the *slope_difference* pre-computation. These errors are due to the fact that *slope_difference* does not always set the weights on the aircraft as it (the weight setting) should be done, and its filtering part may remove important aircraft from the list of possible interval changes.

Parallelism is introduced when searching the 6 element subsets are searched within the 10 potential interval change element sets.

The mean value of P is, however, still close to 120 (storage time maximum value). In order to reduce the number of branches in the search tree, a new labelling, which reduces P, has been implemented.

These 2 new ways of using *min_max* make benchmarking with one *worker* easier without loosing any interesting solutions.

"min_max" vs. "minimize"

The PSAP 3 labelling procedure consists in a interval change sub-list generation followed by interval value setting. Parallelism is introduced in the sub-list generation. The interval values are set by the compulsory *interval direction* labelling procedure.

The trivial way to write this labelling procedure is :

```
min_max(parallel_generation(SL), setting_values(SL))
```

This new labelling procedure does, however, seem more appropriate when using the *minimize* built-in than for the *min_max*. In fact, :

[2] the formula is the combination C_X^Y, but giving an example with a realistic value is the only way to show the improvement from PSAP 2 to PSAP 3

- a priori there is no point in restarting the branch and bound from the top of the search tree where the interval change point subsets are generated,
- it is more efficient to re-start from the bottom of the tree where interval changes and interval values are fixed.

Hence, this labelling procedure has been rewritten :

```
minimize(parallel_generation(SL), setting_values(SL))
```

Tuning the parallel labelling in PSAP 3 consisted in finding with one and several *workers* which amongst the solutions is the best. Benchmarks always showed best running time with the *minimize* built-in use whatever the number of *workers*.

Use of the new *cost_parallel_min_max* and *cost_parallel_minimize* built-ins also has to be studied.

Benchmarks showed running times with the *minimize* built-in use outperform those with the *cost_parallel_min_max* built-in use. The late availability of the *cost_parallel_minimize* built-in has not allowed to benchmarking it.

The PSAP 3 search tree, although it has the same shape as the PSAP 2 search tree, is smaller, the PSAP 3 labelling is more driven by the pre-computation, the PSAP 3 pre-computation is more reliable and more tractable by the end-user : the PSAP 3 optimal solution access time is much more reasonable than the PSAP 2 access time.

New Labelling

The idea of this new labelling consists in reducing the number of possible values for a production interval, P, in a realistic way without removing "good solution" in terms of cost.

This new labelling is implemented as the *interval direction* labelling, i.e. the highest possible production interval is chosen in order to obtain the shortest storage time. But, once this value has been chosen, the following points are added in the new labelling:

- first, store the first value that is compatible with the propagation of the whole set of constraints,
- then, starting from this value in order to try only relevant values, allow other possible $P - 1$ values.

As the production interval first value is the highest possible value and, as the aim is to reduce the storage cost, it appears meaningless to search a solution with a high P value. The writing of the formal proof of this statement has yet to be completed. Once such a formal proof exists, we will be able to say that the optimal solution of the remaining search tree is also the optimal solution of the whole search tree (i.e. the search tree with $P = 90$).

This labelling is hereafter called **new interval direction**.

6.3 PSAP 2 Benchmarking

As PSAP 2 did not prove a successful way to solve the problem, we only present the more meaningful parallel execution benchmarks in order to assess what has been explained in the above PSAP 2 sections.

Moreover, on account of the long running time needed for sequential execution, only a few benchmarks were conducted.

All benchmarks were only done for (sub)optimal cost research.

All benchmarks concern one assembly line schedule, most often the last one, the most difficult to schedule as already said.

The benchmarks consider the time required for labelling, i.e. the time for the parallelized procedure. The labelling time is the running time, starting from the first call for the labelling procedure, that is necessary to prove the (sub)optimality of the solution. It is also called the elapsed time. This labelling procedure takes the longest running time of the PSAP 2 program. The running time to set all constraints and to make the statistical pre-computation is between 20 seconds to 2 minutes for all data sets.

The PSAP 2 benchmarks were not complete for the following reasons :

- they were not run at least 5 times, due to the very long time needed to find solutions with one worker,
- they were not made for all possible numbers of *workers*,
- some of them were made with a greater number of *workers* than available on the used computers (this was done to get an idea on the actually needed parallelism).

The selected benchmarks are those which are most relevant to show the different optimizations (storage time and number of interval changes) and the main drawbacks (search space too big and running time too long) of PSAP 2. They also assess the need for parallelism on account of their speed-ups.

The result of these first benchmarks and the difficulty to achieve them have led to designing PSAP 3 instead of finishing the benchmarks on PSAP 2.

Three benchmarks are presented : the aim of the first benchmark was to minimize the number of interval changes, the aim of the others was to minimize the storage time for a given maximum number of interval changes.

First Benchmarking

They concerned a data set of 198 Mirage 2000.

Statistical pre-computation and *parallel_3_100* labelling were used.

The used branch and bound built-in was the ElipSys *min_max*. The found number of interval changes cost was the optimal solution because the lower bound was set to 1 and

of there was no use of the *X percent better solution than the previous one* in the *min_max* built-in.

The same optimal number of interval changes with the same storage cost was found whatever the number of *workers*.

The following results were obtained on the 12 processors of the Sequent Symmetry computer.

One trial from one to eight *workers* were made and the following array shows for each *number of workers* :

- *labelling time in mn.* is the labelling time to prove the optimality of the number of interval changes,
- *speed-up* is the speed-up between the one *worker* elapsed time and the two, three, ... and eight *workers* elapsed time.

number of workers	1	2	3	4	5	6	7	8
time in mn.	625	234	199	180	166	149	129	122
speed-up	-	2.67	3.14	3.45	3.75	4.19	4.85	5.11

As shown in this array, the speed-ups are quite super linear for 2 and 3 *workers*, then "sub linear" (i.e. 7 *workers* are 5 times as fast as 1). Even if it is interesting to save 8 hours out of 10 hours computation time for 7-8 *workers*, and to save 6.5 hours with 2 *workers*, it is still too long for an interactive tool.

Second Benchmarking

They concerned a data set of 105 Falcon.

The maximum number of interval changes was 5. Statistical pre-computation and *interval direction* labelling were used.

The used branch and bound built-in was the ElipSys *min_max*. The found storage time cost was a suboptimal solution because of the set lower bound and because of the use of the *10 percent better solution than the previous one* in the *min_max* built-in. The lower bound was set to search a solution at least 20 percent better than the first found.

The following results were obtained on the 12 processors of the Sequent Symmetry computer.

One trial for six *workers* and one for one *worker* were made, and the following array shows for each *number of workers* :

- *labelling time in mn.* is the labelling time to prove the suboptimality of the storage cost,
- *first solution (1/2days)* is the first found storage cost in half days,

- *second solution (1/2days)* is the second found storage cost in half days.
- *third solution (1/2days)* is the third found storage cost in half days.

number of workers	6	1
labelling time (mn.)	85.38	more than 20 hours
first solution (1/2days)	2952	2952
second solution (1/2days)	2648	2612
third solution (1/2days)	2298	not found after 20 hours

The same benchmark on the 4 processors of the ICL DRS 6000 computer gave (the legend of this array is the same as the previous one, except that 8 *workers* were used instead of six and that a forth solution was found with 8 *workers*) :

number of workers	8	1
labelling time (mn.)	15.33	more than 12 hours
first solution (1/2days)	4291 (found in few seconds)	2952 (found in few seconds)
second solution(1/2days)	2952 (found in few seconds)	2612 (found in few mn.)
third solution (1/2days)	2648 (found in few mn.)	2323 (found in 12 hours)
forth solution (1/2days)	2207	not found after 12 hours

In the two benchmarks, note that the last solution found with 6 and 8 *workers* was not found with one *worker* and that its search was stopped, hence the suboptimality yet to be proved. Moreover, the second and third solutions were not the same with one *worker* as with several *workers*. This is due to parallel execution which explores branches in the search tree which are different from those searched in sequential execution.

The speed-ups on Symmetry and on DRS 6000 are super linear, since the result with one *worker* respectively was at least 10 and 44 times as slow than with 6 and 8 *workers*. We say *at least* because the last solution was not found and the suboptimality not proved.

Moreover, if we consider the quality of the solution, 8 *workers* on DRS 6000 found a better solution than 6 on Symmetry. It may not be meaningful to compare DRS 6000 and Symmetry results, but it is clear that more parallelism improves the speed-ups and explores a more fruitful part of the search space in the alloted time.

Third Benchmarking

It concerns a data set of 130 Mirage 2000. For once, it involved the second assembly line (used only for the manufacture of single seater aircraft) where 70 aircraft are manufactured. This line is as difficult to schedule as the last assembly line because the following line manufactures both single and twin seater aircraft.

The maximum number of interval changes was 3. *Interval direction* labelling was used but not statistical pre-computation.

The used branch and bound built-in was the ElipSys *min_max*. The found storage time cost was a suboptimal solution because of the set lower bound and because of the use of

the *10 percent better solution than the previous one* in the *min_max* built-in. The lower bound was set to search a solution at least 20 percent better than the first found.

The following results were obtained on the 4 processors of the DRS 6000 computer. One trial for eight *workers* and one for one *worker* were made, and the following array shows for each *number of workers* :

- *labelling time in sec.* is the labelling time to prove the suboptimality of the storage cost,
- *first solution (2842 half days)* is the elapsed time (in seconds) needed to find the first found storage cost (2842),
- *second solution (2548 half days)* is the elapsed time (in seconds) needed to find the second found storage cost (2548).
- *third solution (2248 half days)* is the elapsed time (in seconds) needed to find the third found storage cost (2248).
- *forth solution (2012 half days)* is the elapsed time (in seconds) needed to find the forth found storage cost (2012), which is the suboptimal cost.

number of workers	8	1
labelling time (sec.)	98	340
first solution (2842 half days)	8 sec.	1 sec.
second solution(2548 half days)	5 sec.	2 sec.
third solution (2248 half days)	7 sec.	2 sec.
forth solution (2012 half days)	10 sec.	256 sec.

8 *workers* were 25 times faster than one in reaching the 71 percent better solution (2012 half days) and 3 times faster to prove its optimality. In this case, the same storage cost were found with eight and one worker.

Conclusions

Benchmarks with greater size data sets are impossible because the running time takes more than 10 hours with one *worker* and more than one hour with 4 or 8 *workers*. However, the promising speed-ups found with these two small data sets led us to safely prune the search tree in such a way that parallelism would give results in an acceptable running time. PSAP 3 was the way chosen to prune the search tree.

Moreover, *ParSee* analysis showed :

- no parallelism overhead in PSAP 2 (e.g. due to communication between ElipSys *workers*),
- all the *workers* are well used,
- 30 *workers* would be used by PSAP 2.

At that time, it would have been worth having access to a 30-processor computer to check this forecast and look at the speed-ups.

6.4 PSAP 3 Benchmarking

Numerous benchmarks were conducted. In our search to find the bounds of the search tree manageable by PSAP 3, three classes of benchmarks emerged :

- benchmarks on a small search tree whose speed-ups given by parallel execution are *sub linear*, i.e. for N *workers* the speed-ups are smaller than N,
- benchmarks on a large search tree but with small grain size whose speed-ups given by parallel execution are *sub linear*,
- benchmarks on a large search tree but with large grain size whose speed-ups given by parallel execution are *super linear*, i.e. for N *workers* the speed-ups are greater than N.

One representative benchmark of each classes is presented hereafter.

All benchmarks were only done for (sub)optimal cost research.

All benchmarks concerned one assembly line schedule, it was always the last one, the most difficult to schedule as already said.

As for PSAP 2, the benchmarks consider the elapsed time to label, i.e. the time for the parallelized procedure. But, the PSAP 3 labelling procedure consists in a interval change sub-list generation followed by interval values setting. Parallelism was introduced into the sub-lists generation. The interval values are set by the compulsory *new interval direction.*

This labelling procedure takes the longest running time of the PSAP 3 program. The elapsed time to set all constraints and to make the statistical pre-computation is between 20 seconds to 2 minutes for all data sets.

Their aim was to minimize the storage time for a given maximum number of interval changes. They were made on the 4 processors of the ICL DRS 6000 computer.

The benchmarks were made on the first PSAP 3 optimization. This optimization searches Y production interval changes out of X possible aircraft. The X aircraft are given by the *slope_difference* pre-computation.

The first task done was to define X and Y for each data set. As Y is a subjective parameter given by the planner, it remained to be defined which values of X, for a given Y and a given data set, would give an acceptable behaviour (i.e. acceptable running time and acceptable solution quality) in sequential execution and speed-ups in parallel execution.

For a sequential execution, C_X^Y must not be too big, for a parallel execution $X - Y$ must not be too small. As a matter of fact, C_X^Y gives an idea of the search tree size, and to obtain improvements from parallel execution, this size must not be too small. Several benchmarks with an X value from $Y + 1$ to 10 for the small-size data sets or up to 20 for

the large-size data sets have shown some bounds to the value of C_X^Y. The conclusion of this section will give these bounds.

When X is chosen for a given data set, the solution's quality must be checked by the planner. The graphic interface shows him the different X change possibilities. He can check that no relevant aircraft is removed from the interval change possibilities list.

The first two presented benchmarks are the lower and upper bound of the actual data set size. The last one is a medium size data set.

The first benchmark concerned a small data set and search for the optimal storage cost.

The second benchmark concerned a large data set and search for a suboptimal storage cost.

The third benchmark concerned a medium size data set and search for the optimal storage cost.

First Benchmarking

A data set of 70 Mirage 2000 was chosen.

The maximum number of interval changes was 3. Slope_difference pre-computation and *new interval direction* labelling were used.

The chosen optimization was to choose 3 production interval changes out of 10. The search tree size hence was the combination C_{10}^3, i.e. there were 120 subsets with 3 changes to be explored.

The used branch and bound built-in was the ElipSys *minimize*. The found storage time cost was the optimal solution because the percent parameter of *minimize* was not used and the lower bound set in the *minimize* built-in was smaller than the found storage cost (1199 half days for the lower bound, 1636 half days for the optimal cost).

Whatever the number of *workers*, two storage time values were found before the optimal storage time value.

8 trials for each number of *workers* were made, and the following array shows two means for each *number of workers* :

- *time in sec.* is the arithmetical mean of the eight elapsed times, given in seconds,
- *speed-up* is the geometrical mean of the eight speed-ups between the one *worker* elapsed time and the two, three and four *workers* elapsed time.

number of workers	1	2	3	4
time in sec.	534	280	198	160
speed-up	-	1.9	2.69	3.32

Second Benchmarking

A data set of 250 Mirage 2000 was chosen.

The maximum number of interval changes was 11. Slope_difference pre-computation and *new interval direction* labelling were used.

The chosen optimization was to choose 11 production interval changes out of 14. The search tree size hence was the combination C_{14}^{11}, i.e. there were 364 subsets with 11 possible changes to be explored.

The used branch and bound built-in was the ElipSys *minimize*. The found storage time cost was a suboptimal solution as (although if the percent parameter of *minimize* was not used) the lower bound set in the *minimize* built-in was greater than the found storage cost (7000 half days for the lower bound, 6627 half days for the optimal cost). Besides, only 84 subsets were explored instead of the 364 possible subsets.

With one worker, five storage time values were found before the optimal storage time value.

With two, three and four *workers*, four storage time values were found before the optimal storage time value.

8 trials for each number of *workers* were made, and the following array shows two means for each *number of workers* :

- *time in mn.* is the arithmetical mean of the eight elapsed times, given in minutes,
- *speed-up* is the geometrical mean of the eight speed-ups between the one *worker* elapsed time and the two, three and four *workers* elapsed time.

number of workers	1	2	3	4
time in mn.	212	52	22	18
speed-up		4.02	9.7	11.51

This optimal storage time only needs 10 interval changes which means that one possible changes was removed by the good behaviour of *new interval direction* labelling.

Third Benchmarking

A data set of 99 Falcon was chosen.

The maximum number of interval changes was 3. Slope_difference pre-computation and *new interval direction* labelling were used.

The chosen optimization was to choose 3 production interval changes out of 11. The search tree size hence was the combination C_{11}^{3}, i.e. there were 165 subsets with 3 changes to be explored.

The used branch and bound built-in was the ElipSys *minimize*. The found storage time cost was the optimal solution because the percent parameter of *minimize* was not used and the lower bound set in the *minimize* built-in was smaller than the found storage cost (3000 half days for the lower bound, 3122 half days for the optimal cost).

Whatever the number of *workers*, the first storage time value found was the optimal storage time value, except sometimes with 4 *workers*.

Ten trials for each number of *workers* were made, and the following array shows two means for each *number of workers* :

- *time in sec.* is the arithmetical mean of the ten elapsed times, given in seconds,
- *speed-up* is the geometrical mean of the ten speed-ups between the one *worker* elapsed time and the two, three and four *workers* elapsed time.

number of workers	1	2	3	4
time in sec.	77	43	32	33
speed-up		1.79	2.39	2.34

The speed-ups were smaller here than for the first PSAP 3 benchmark because the optimal cost was in one of the first left branches of the search tree and the 3 interval changes constraint propagation thus became very efficient and reduced the parallel grain size in such a way that the communication overhead slowed down the parallel execution.

Conclusions

This conclusion discusses the reasons for such speed-ups differences between the second benchmarking and the others. The object is to find the bounds within which PSAP 3 benefits from parallel execution.

From these two benchmarks, and from others benchmarks not presented here, the search tree size manageable by PSAP 3 can be discussed. This size depends on the value of C_X^Y, on the size of the parallel grain and on the data set size.

The following search space bounds were found for an optimal storage cost search with **four workers** :

- C_X^Y smaller than 100, whatever data set size smaller than 100 aircraft :
  parallel execution provides no meaningful improvement, as running times are below ten minutes with one *worker*. Although in the first benchmarking, it is worth to get solutions in less than 3 minutes with four *workers*, the search tree size is not enough large to require parallel execution. As can be seen in the first benchmarking above, speed-ups are *sub linear*, i.e. for N *workers* the speed-ups are smaller than N. This condition means also that it is not worth trying $X \leq Y+2$, as the planner will never request more than 10 for X[3] (for such small-size data sets).
- whatever C_X^Y, whatever data set size smaller than 100 aircraft :
  if constraint propagation reduces the parallel grain size too much, the speed-ups become non-existent. This case may also occur for data set sizes greater than 100 aircraft. The problem here is to know a priori the parallel grain size. This size depends on the number of interval changes, on the data set size and on the first solution found. The following remarks can be made :

[3] $C_{10}^8 = 45$

- if the number of interval changes is smaller than 4, a small parallel grain sizes are possible, whatever data set size,
- small parallel grain sizes mainly occur for small data set sizes which are not the actual data set sizes of PSAP,
- if the first solution found in sequential execution is the optimal solution, the parallel grain size may be too small to obtain speed-ups while searching for the solution optimality proof if there is good constraint propagation.

- C_X^Y greater than 100, whatever data set size greater than 150 aircraft :
  parallel execution with 4 *workers* cannot handle such a large search tree in an acceptable time . First trials with more than 7 *workers* on a 12-processor SGI computer showed that such large search trees can be handled in an acceptable time.

This last point concerns the actual data sets, those which really require parallel execution. PSAP 3 was modelized to find solutions for actual data sets on 12-processor computer and it does.

The search space bounds may be studied for an suboptimal storage cost search with **four workers** (cf. second benchmarking). There, the suboptimal cost was found in the first 84 subsets studied out of the 364 subset possibilities.

To give an idea of the time needed to prove solution optimality in the case of the second benchmarking, the *parallel_generation* procedure was reversed in order to generate the subsets in the reverse order, and the same benchmark gave, with one *worker* on DRS 6000, the same suboptimal storage cost[4] after 7 hours and 16 minutes' running time instead of the previous 212 minutes (i.e. 3 hours and 32 minutes). This running time is too long to conduct benchmarking with four *workers*.

PSAP 3 parallel execution for actual data sets improves the running time to prove the storage time cost optimality or to find a suboptimal storage time cost. In its current state, PSAP 3 needs more than seven *workers* to provide a running time of less than five minutes for a suboptimal search. More benchmarks are needed with the 12-processor SGI computer to determine the running time to prove a cost optimality.

6.5 Interaction between Parallel and Sequential Improvements

The purpose of the sequential improvements introduced in PSAP 2 mainly are to put the suboptimal solutions in the left part of the search tree but the search tree remains too large to obtain the remaining search tree optimal solution.

The sequential improvements introduced in PSAP 3 thus are mainly aimed to reduce the search tree in order to obtain the remaining search tree optimal solution.

The sequential improvements introduced in PSAP 3 result in poor speed-up in parallel for the small-size data sets (e.g. 3 times faster with 4 *workers* instead of dramatic *super linear* speed-up in PSAP 2).

[4]this suboptimal cost thus is the optimal cost

For the actual data sets, *super linear* speed-up are even more important for PSAP 3 than for PSAP 2. This is mainly due to the PSAP 3 search tree pruning provided by *new interval direction* labelling.

Nevertheless, for all the data sets, the access time has been improved, i.e. whatever the number of *workers* between 1 and 4, the access time for a given number of *workers* is much better in PSAP 3 than in PSAP 2.

In terms of access time, the sequential improvements has improved both sequential and parallel behaviour of PSAP 3.

7 Conclusions

7.1 Parallel CLP Assessment

PSAP belongs to a mixed scheduling and planning application area : scheduling area by its precedence constraints and planning area by the pacing problem. The main characteristics of PSAP application area can be defined by the PSAP specifications :

- the precedence and interval production constraints,
- two cost criteria, they are the storage time and the workload,
- the compromise between these two costs depends on the current production policy, moreover there is no function to optimize these two costs,
- the planners do not use any algorithm but their expertise to pace the assembly lines with respects to the current production policy about the resulting costs.

The given compromise between the two costs is to give a maximum number of interval changes and to let PSAP searching for the (sub)optimal storage time cost.

Given the simplified optimality search, our initial intuitions were that :

- CLP could express in an easy and declarative way the specified constraints but these constraints do not much propagate on the domain variables, this leads to an infeasible optimality search,
- parallel execution could help to search the optimal (or at least suboptimal) cost into the large remaining search tree.

These initial intuitions were verified and super linear speed-ups were find. But they were too simplistic, as PSAP 2 *ParSee* analysis showed : **more parallelism** than currently available was needed. This is because of the weakness of propagation between the different constraints implies a too huge search tree.

Considering PSAP 3 results, the use of parallel CLP and its analysis tool, *ParSee*, is well suited for PSAP, as it allows :

- to know the search tree size and to give ideas to reduce it to a size manageable by the current ECRC's computers,
- to tackle previously infeasible problem as:
  - the optimality proof search for small data sets,
  - to find, for actual data sets, suboptimal costs which are actually better than costs found with a sequential execution. And it may be possible to prove cost optimality in an acceptable running time.

Characteristics for applications in the same area than PSAP can be :

- some constraints which the propagation does not prune a lot the search tree,
- a function for the cost criteria which the propagation does not prune a lot the search tree,
- a lot of man expertise to assess the modelization, pre-computation and heuristics chosen to prune the search tree,
- a need of an interactive tool used by the experts,
- a need to quickly optimize the solution quality and costs,
- a sequential execution of an optimal cost search takes several hours.

If an application has those characteristics, parallel CLP is adequate as it was for PSAP.

7.2 Enhancements and Extensions to Parallel CLP System

Most of the enhancements and extensions to parallel CLP system, useful for PSAP, are already included in ECL^{i}PS^e. Some of them are the cost parallel branch and bound built-ins : *cp_min_max* and *cp_minimize*. Another is to allow the use of more *workers* : by the port to new multi processors platforms as the 12-processor SGI or, when super linear speed-ups are expected to allow the use of more *workers* than available processors.

Some useful enhancements concern rather the ECL^{i}PS^{e}debugging and analysis tools.

When a parallel execution takes too long it could be useful to stop it and to switch into a debug mode to check if :

- all the processors are actually active,
- they are doing more communications than processing.
- there is no loop, or any other bugs, because this can occur in a branch of the search tree never reached by a sequential execution.

ParSee analysis was the key point to understand the PSAP 2 performance, it will be nice to provide it with ECL^{i}PS^e. As it can predict the ability of thirty processors use from a four processors execution, it can also predict parallel grain size or parallelism need from a sequential execution.

Another analysis tool, could also give an idea of the constraint propagation size in terms of elapsed time and in terms of search tree branch pruning.

7.3 Development Total Effort

It is difficult to estimate the total effort required for developing PSAP :

- as several engineers and last year students contributed to the development which started before the APPLAUSE project. Most of the persons, with PROLOG background, learnt CLP then parallel CLP developing PSAP which is not the best way to learn a new technology. But most of these students and all the engineers became efficient within two months, which is a good indication of the declarativity of the technology and to its accessibility by new users.
- Before the APPLAUSE project beginning, PSAP was stopped then restarted.
- Moreover, the Argenteuil's planner changed and some PSAP specifications changed with them.

To reach PSAP 3 and its user-friendly interface, more than 5 man-years were required without the specifications time. From today, such a system will need one man-year for the specifications and 3 man-years of development effort.

The difficulty encountered in the parallelization process was very small. As a matter of fact, most of the time lost was due to difficulties with temporary problems of the pre release of ElipSys and ECL^{i}PS^e. The time spent for introducing parallelization is anyway significantly less than for each of the other phases : specification, modelling, expression of the constraint, sequential improvements. The time to adjust the search tree to a size manageable by the provided parallel computers was the time to improve the sequential behaviour. This last time was the most important since this means several trials of different pre-computations, heuristics and optimality searches.

7.4 Other Conclusions

CLP in sequential execution has proved adequate in the search for good solutions, since efficient (though simple) pre-computation with a graphic user-friendly interface could be designed.

PSAP has induced the development in the ElipSys and ECL^{i}PS^{e}system of some general purpose features, like *time/4* and adaptation of the *cost parallel branch and bound* to the *minimize* built-in.

ECL^{i}PS^{e}cannot be applied for the moment to the practical PSAP application because of factors which have nothing to do with parallelism as :

- a formal proof of the **new interval direction** labelling to convince the planner of the actual optimality of the solution,
- an idea in French Francs of the savings obtained by (sub) optimal storage costs with regard to current storage costs in the case of an identical number of changes and of a similar workload,
- the availability of ECL^{i}PS^e on a network of workstations with the same performances as those seen on the SGI computer.

The main conclusions are :

- Parallel execution has made possible to produce a better quality solution with super-linear speed-ups,
- parallel resources when available, can be efficiently and relatively easily exploited in ECL^{i}PS^e for this kind of application.

7.5 Acknowledgments

Let the ECL^{i}PS^e team at ECRC be thanked for their highly qualified and friendly help. At Dassault Aviation, Patrick Albers, André Chamard, Marc Sicard and Annie Fischler have worked on the internal application together with the author of this report; this work could not have been carried out without their efficient participation. The PSAP 3 benchmarks were mainly done by Patrick Albers (LAAS-CNRS) and by Shyam Mudambi (ECRC). Thanks also to Vincent Sarracanie who did hard work on this problem and its complexity during its stay at Dassault. Last but not least, thanks to Ute Nichols for her patience to correct my poor English in almost all this report and some others.

Chapter 4.

The TCO Application

André Chamard

1 Introduction

The training of aircraft pilots is an important issue for an aircraft manufacturer. This has to be taken into account from the initial design until the commercial phase. It has indeed become impossible to sell an aircraft without consideration for the appropriate training system. This is the reason why an internal study on pilot training has been launched at Dassault Aviation. Increasing cost and complexity of the training process, indeed, have made it necessary to re-consider the whole training curriculum. This implies assessing the relative capabilities of the training means, taking into consideration the fast progress of simulation and estimating the potential benefits from an earlier training for some of the piloting tasks.

Pilot training has been modelled by operational experts at Dassault Aviation. The perspective is now to develop a decision support system based on the model they have elaborated. This task has been assigned to the *Artificial Intelligence and Advanced Computer Techniques Department* of the *Advanced Studies Division*. A prototype has been implemented in Constraint Logic Programming (CLP), benefiting from the Department's previous experience with these techniques, in particular for production management problems [BCP92, CF94, CF95, BCF95, CFGG95a, CFGG95b]. Parallel CLP has been identified as a potential way of overcoming certain performance limitations of sequential CLP due to the highly combinatorial nature of the problem. Therefore the TCO application has been chosen as a contribution of Dassault Aviation to APPLAUSE, and a parallel version of the prototype has been developed and systematically tested for the project.

This report provides a description of the problem, of its modelling and the strategies for solving it, both in sequential and parallel CLP (ECL^iPS^e). The approach adopted here could be applied to any training problem involving a variety of training means with significantly different efficiencies and costs. The conclusions drawn do not seem, therefore, to be limited to this particular training optimization problem.

2 Problem Description

The Context

Pilot Training Pilot training is a long and selective process. The pilot students' curriculum consists of several phases, each characterized by the location at which they take place and the type of training. They can be thought of as school years, even though their durations may not be a year. The number of pilot students decreases at each phase. There are currently five phases: *Primary, Basic, Advanced, OTU (Operational Training Unit), OCU (Operational Combat Unit).*

The whole range of piloting competences the pilot should have acquired at the end of the curriculum is divided into what is called *piloting tasks*. *Taking Off*, *Landing*, *Night Flying*, *Patrol Flying*, *Low Altitude Flying* are examples of tasks. This division depends on the level of granularity at which the curriculum is considered: *Low Altitude Flying*, for instance, can be further decomposed into *Low Altitude Flying with Visibility* and *Radar-Based Low Altitude Flying*. For curriculum planning purposes, however, it is useless to

go into much detail. 22 tasks have been identified as a basis for this study. For each task, the pilot has to reach a required level, referred to by convention as 100%, at the end of the curriculum. The tasks are usually taught in more than one phase, often in all of them. Continuity is important. The teaching of *Acrobatic Flying* eg. currently starts in the *Primary* phase on small planes, continues in the subsequent phases and is finally completed on the operational combat aircraft (currently, the Mirage 2000).

Training is performed on training means available in the training schools and air-force bases: these are planes, of course, ranging from small training planes to two-seater and single-seater combat aircraft, as well as a variety of simulator types, from simple PC simulator programs to complete mission simulators, including cockpit trainers, partial task trainers, etc. Their cost dramatically increases with the level of sophistication and realism. Simulators may have a cost greater than that of the small training planes.

Current Trends The level of performance required from modern aircraft has a strong influence on pilot training at least in two respects:

- They are more and more expensive. This has a direct impact on the cost of training, since combat aircraft have to be used for the training. The overall cost is already of the order of two million Ecus per pilot. It will increase dramatically if the current training structure is kept.
- The systems are more and more complex. This makes the pilot's tasks harder and harder and the tendency is towards a longer training.

The design of the future pilot-training schemes has to take these evolutions into account.

Designing a Training System. The TCO Project The training means cannot be considered in isolation. A global view of the training system is necessary if one wants to use the means in such a way that the cost is kept down at a reasonable level. The aim of the TCO (Training Curriculum Optimization) project is to allow to design optimal curricula given a set of means considered as available. This availability may be real, in the perspective of a short-term improvement of the current practice, or potential, in a more prospective approach.

A Learning Model

The Basic Learning Model Comparing the characteristics of means is a key issue, which requires choosing a learning model. Most training processes can be represented by 'sigmoid' curves (Figure 2.1): early progress is slow, then learning proceeds at a constant speed until some asymptotic level is approached.

This kind of model has been adopted in TCO for the training to a particular task on a given means. But this model is basically non-linear and would lead to great computational difficulties in an optimization perspective if it were kept as such. The curve, however, can be simply characterized by three parameters: a *familiarization (adaptation) time* (f), an

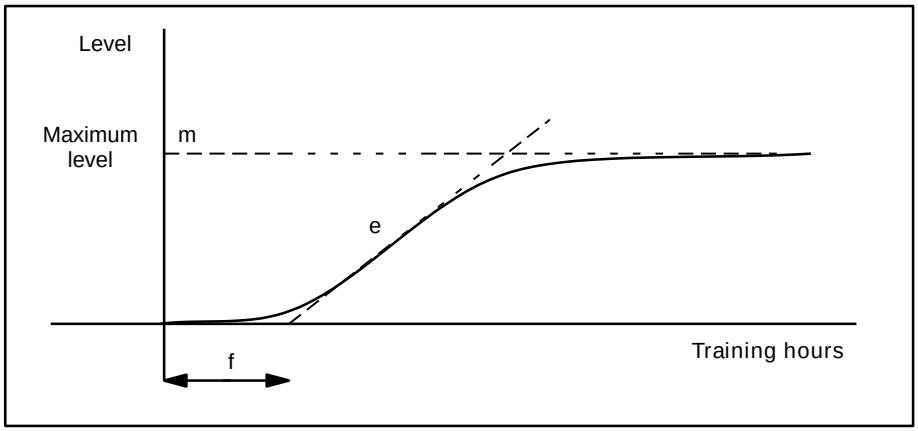


Figure 2.1: Learning Curve

efficiency (e), which is the learning speed, and a *maximum reachable level* (m) — in short, *maximum level* or *ceiling*. In practice, the learning curve has been considered (by the operational experts themselves) as piecewise linear: first, the familiarization (adaptation) time during which no progress is achieved, then a constant learning speed, until the ceiling is reached. This kind of law is much more tractable, with no significant alteration of the model's significance.

Reaching the Required Competence Level for a Task Learning a task is achieved by using a set of means. For instance, the task *Normal Domain Flying* was decomposed in 1992 as follows:

PHASE #	PHASE NAME	TRAINING
1	Primary	5 hours on Epsilon
2	Basic	25 hours on Alpha Jet
3	Advanced	4 hours on mission simulator
4	OTU	no training for this task
5	OCU	3 hours on complete simulator, 3 hours on two-seater combat aircraft, 3 hours on single-seater combat aircraft

This leads to the kind of representation given in Figure 2.2.

The temporal aspect is expressed by the phases. In the model adopted here there is no notion of chronology inside a phase: the means are simply sorted by increasing order of their maximum levels. How their utilization by all the students will be eventually scheduled is not relevant for the TCO project. TCO is about planning the curriculum and is not concerned with scheduling nor time-tabling.

It is important to note that a means with a relatively low maximum level cannot be used if its ceiling has been already overstepped by a more powerful means in an earlier phase. This is a crucial point which will be discussed in detail later. For the model to be complete, a notion of maturing should also be introduced: the learning speed in a given phase also depends on the level reached previously; the higher the already acquired competence (in

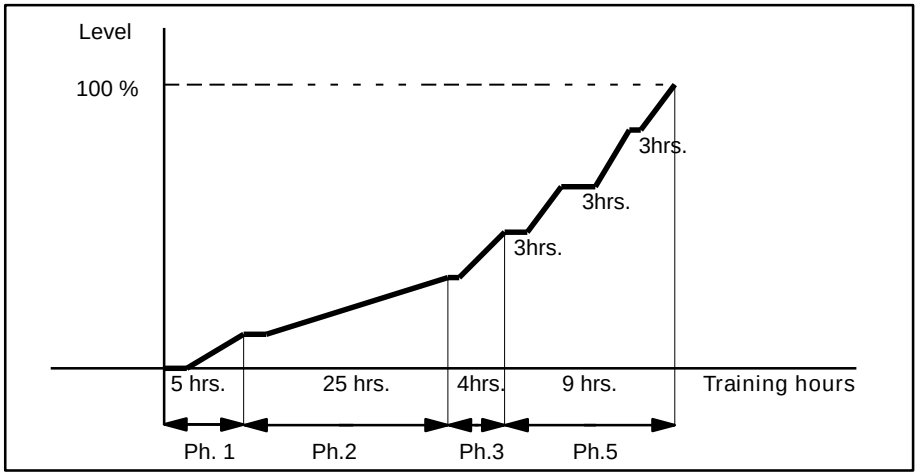


Figure 2.2: The Normal Domain Flying Task

previous phases, with other means), the higher the speed. This presentation of the chosen learning model allows us now to define the problem in a more complete way.

Problem Description

Data and Results A tasks decomposition is assumed, as well as the declaration of the means available in the phases. Defining the curriculum amounts to determining the number of training hours (possibly a non integer number) assigned to each means for each task in each phase. This provides a kind of teaching programme, comparable to those applicable to standard education at the state level. They define the number of hours to be dedicated to the various activities and lessons, for each school year and each subject, and the levels to be reached. Time-tabling is left up to the teachers' and headmasters' organization skills.

The basic data are therefore the following:

- a list of tasks
- a list of phases
- a list of means declared as available for each task and each phase
- the utilization cost per hour of each means
- the maximum reachable level for each means declared as available in a phase for a task
- the relative efficiencies of the means (in the different phases and tasks) compared to that of the two-seater combat aircraft
- the utilization times of the means in the current curriculum (which will allow to compute absolute efficiencies).

Constraints of pedagogical or organizational nature are specified:

- total duration of the curriculum
- total duration for certain phases
- minimum or maximum utilization of certain means
- minimum number of flying hours
- minimum level to be reached at the end of certain phases for certain tasks
- etc.

The results are

- for each phase and each task, the utilization time of all the available means.

The objective is in general to minimize the total cost. But one may also want to compute the theoretical minimum duration (ie. with no cost constraint). The cost minimum with no duration constraints on the one hand, the duration minimum with no cost constraint on the other hand are nothing but two extreme points of a $Cost = f(Duration)$ curve which is quite meaningful for the users, and on which they intend to perform sensitivity analyses. The most acceptable compromise between cost and duration is likely to be in some area of the curve where the cost is relatively low, but a significant duration decrease can be achieved by just slightly increasing the cost (Figure 2.3).

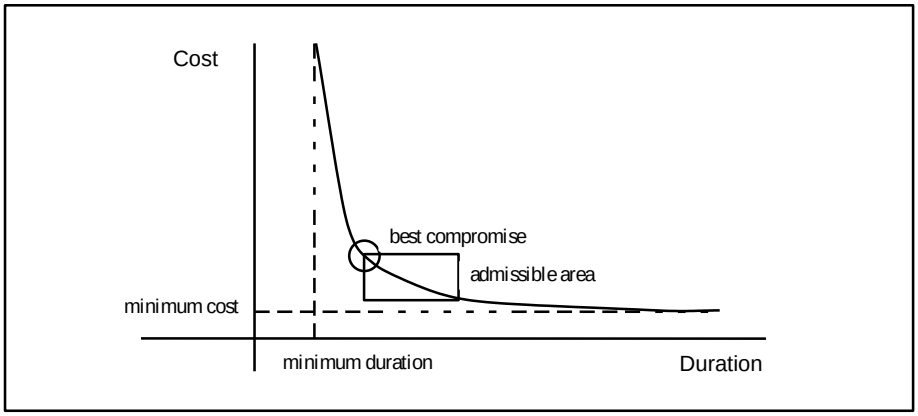


Figure 2.3: Cost–Duration Curve

A Global Problem Constraints such as maximum duration for the whole curriculum or for a phase, or minimum number of flying hours, involve all the tasks. This makes it impossible to split the problem by optimizing the tasks separately. As a consequence, a full problem with the current data does correspond to 22 tasks and 5 phases.

Origin of the Data Maximum reachable levels and relative efficiencies are parameters whose necessity is implied by the learning model. But they are not normally handled by domain experts and are not accessible through immediate experience. A reflection on the procedures to be set up in order to provide experimental criteria for the comparison of the

means is under way. The current data on the tasks have been so far defined by one expert. They are consistent but more based on intuition than on observable criteria. Yet they are sufficient for the purpose of testing the model and its computer implementation. In any case, the issue of parameter acquisition and validity, though crucial for the TCO project, is outside the scope of APPLAUSE and will not be discussed further in this report.

Approximate Optima The retained learning model, being piecewise linear, has a certain degree of approximation. It is then sensible that the requirements should not be for exact optima, but for approximate ones. The practical objective is therefore to find solutions within a given percentage (5% eg.) of the theoretical (but not totally meaningful) optimum. It is in fact expected that the computer system will be able to produce several curricula, whenever possible, within a specified percentage of the global optimum.

3 Problem Qualification

A Mixed Integer-Linear Problem

Sketch of the Problem Structure In the rest of this document, the following notations will be adopted:

- the means are indexed in chronological order of the phases and in order of increasing maximum levels inside each phase
- the utilization time of means m_i is T_i, expressed in hours
- its cost per hour is c_i
- $P_i = e_i * T_i$ denotes the (possibly null) progression achieved with m_i, where P_i is expressed in percents of the required level and e_i is the efficiency of m_i, in percents of level per hour
- $L_i = P_1 + ... + P_{i-1} + P_i$ is the level reached after the (possibly null) utilization of m_i.

For the purpose of explanation, a simplified problem featuring a single task will be considered now. It will be assumed that 4 means are declared available, over 2 phases: the first two means can be used in Phase 1, the other two ones in Phase 2. It will be assumed as well that the familiarization times are negligible. The means' characteristics are given below:

Phase	Means	Efficiency	Maximum Level	Cost
1	m_1	4	40%	1
	m_2	8	60%	5
2	m_3	2	50%	2
	m_4	10	100%	20

The corresponding set of equations will express that

- the level reached just after actually using a means cannot be higher than its maximum reachable level; if the means is not used, then no constraint is imposed
- the level reached at the end should be 100%
- the overall cost is the sum of the utilization times multiplied by the costs per hour.

The equations are the following (with conjunction between equations noted by commas):

$MINIMIZE\ Cost = 1 * T_1 + 5 * T_2 + 2 * T_3 + 20 * T_4$
$UNDER$
$T_1 \geq 0,\ P_1 \geq 0,\ L_1 \geq 0,$
$T_2 \geq 0,\ P_2 \geq 0,\ L_2 \geq 0,$
$T_3 \geq 0,\ P_3 \geq 0,\ L_3 \geq 0,$
$T_4 \geq 0,\ P_4 \geq 0,\ L_4 \geq 0,$
$Cost \geq 0,$
$P_1 = 4 * T_1,\ L_1 = P_1,$
$P_2 = 8 * T_2,\ L_2 = L_1 + P_2,$
$P_3 = 2 * T_3,\ L_3 = L_2 + P_3,$
$P_4 = 10 * T_4,\ L_4 = L_3 + P_4,$
$(L_1 \leq 40 \vee P_1 = 0),$
$(L_2 \leq 60 \vee P_2 = 0),$
$(L_3 \leq 50 \vee P_3 = 0),$
$L_4 = 100$

The basic constraints are linear and it can be assumed that the variables are continuous. A priori, this kind of problem seems to be within the scope of classical linear programming methods, like the Simplex algorithm. This small example, however, already shows the existence of disjunctions which will deeply affect the problem's complexity.

Several Sources of Disjunctions a) Maximum Level

The above example features three disjunctions related to maximum reachable levels. Only the third one, actually, is a real disjunction, since for means m_1 and m_2 the maximum reachable levels cannot be overstepped by previous ones. Therefore, the associated constraints are also true if these means are not used. Only m_3 is such that a level greater than its own maximum (50%) may be reached by a preceding means (m_2 with ceiling 60%). This allows us to write the following, equivalent but simpler, system:

$MINIMIZE\ Cost = 1 * T_1 + 5 * T_2 + 2 * T_3 + 20 * T_4$
$UNDER$
$T_1 \geq 0,\ P_1 \geq 0,\ L_1 \geq 0,$
$T_2 \geq 0,\ P_2 \geq 0,\ L_2 \geq 0,$
$T_3 \geq 0,\ P_3 \geq 0,\ L_3 \geq 0,$
$T_4 \geq 0,\ P_4 \geq 0,\ L_4 \geq 0,$
$Cost \geq 0,$
$P_1 = 4 * T_1,\ L_1 = P_1,$

$P_2 = 8 * T_2,\ L_2 = L_1 + P_2,$
$P_3 = 2 * T_3,\ L_3 = L_2 + P_3,$
$P_4 = 10 * T_4,\ L_4 = L_3 + P_4,$
$L_1 \leq 40,$
$L_2 \leq 60,$
$(L_3 \leq 50 \vee P_3 = 0),$
$L_4 = 100$

The remaining disjunction cannot be removed. Instead of a conjunction of constraints, which would define a convex polyhedron in the solution space (the intersection of the half-spaces defined by the inequality constraints), the disjunction leads to a non convex polyhedron. The Simplex method is no longer applicable, and the choice of particular branches needs to be made. Moreover, if this simple example features only one disjunction, real problems will show a great number of them. **With respect to the maximum reachable level constraints, a disjunction will be encountered each time a means has a maximum level lower than that of at least a means available in a preceding phase. Such means will be called 'disjunctive means'**. Since the means are sorted by increasing ceilings inside each phase, a means can be disjunctive only because of the presence of a higher-level one in a preceding phase, not in the same phase.

Other sources of disjunction exist, which will now be explained.

b) Familiarization

Familiarization times T'_1, T'_2, T'_3, T'_4 can be added to the above example. These times lead per definition to no progress, but they have a cost. Moreover, one needs to stipulate that these new variables may only take two values: either 0 (if the means is not used), or a given fixed value (respectively, F_1, F_2, F_3, F_4). The new constraint system is the following:

$MINIMIZE\ Cost = 1 * (T_1 + T'_1) + 5 * (T_2 + T'_2) + 2 * (T_3 + T'_3) + 20 * (T_4 + T'_4)$
$UNDER$
$T_1 \geq 0,\ T'_1 \geq 0,\ P_1 \geq 0,\ L_1 \geq 0,$
$T_2 \geq 0,\ T'_2 \geq 0,\ P_2 \geq 0,\ L_2 \geq 0,$
$T_3 \geq 0,\ T'_3 \geq 0,\ P_3 \geq 0,\ L_3 \geq 0,$
$T_4 \geq 0,\ T'_4 \geq 0,\ P_4 \geq 0,\ L_4 \geq 0,$
$Cost \geq 0,$
$P_1 = 4 * T_1,\ L_1 = P_1,$
$P_2 = 8 * T_2,\ L_2 = L_1 + P_2,$
$P_3 = 2 * T_3,\ L_3 = L_2 + P_3,$
$P_4 = 10 * T_4,\ L_4 = L_3 + P_4,$
$((T_1 > 0,\ L_1 \leq 40,\ T'_1 = F_1) \vee (T_1 = T'_1 = 0)),$
$((T_2 > 0,\ L_2 \leq 60,\ T'_2 = F_2) \vee (T_2 = T'_2 = 0)),$
$((T_3 > 0,\ L_3 \leq 50,\ T'_3 = F_3) \vee (T_3 = T'_3 = 0)),$
$T'_4 = F_4,\ \%\ m_4\ has\ to\ be\ used$
$L_4 = 100$

These new disjunctions make the problem even more complex. In addition, it has been assumed that the familiarization time, whenever applied, is fixed. It would certainly be more realistic to make it a function of the history of the learning process: if eg. a pilot

student has already flown on a two-seater combat aircraft, the adaptation time to the single-seater will be almost negligible. It is obviously not the case if he has only piloted training planes. Taking this into account would introduce further difficulties both for modelling and resolution.

c) Maturing

Modelling the maturing process is another hard issue. One may consider defining a maturing factor, by which the means' efficiencies in a given phase will be multiplied. This factor would be a function of the amount of training already carried out eg. at the end of the previous phase (expressed as a duration or a level). In order to keep the basic constraints linear, one would have to introduce thresholds. Durations or levels previously reached being only known a posteriori, this would lead to additional combinatorics.

A Mixed Integer (Binary)-Linear Problem This altogether makes the problem a Mixed Integer (Binary)-Linear Problem. It is binary since all choices can be expressed by boolean variables: pieces of equations like $T_1' = F_1 \vee T_1' = 0$ can, indeed, be replaced by $T_1' = F_1 * B1$, $(B1 = 1 \vee B1 = 0)$. Finding feasible (not necessarily optimal) solutions may imply exploring somehow all binary choices corresponding to utilization or non utilization of the disjunctive means (plus, with more complex models, checking whether certain thresholds are reached). Finding the optimum will imply traversing at least implicitly the whole search space. The time required to solve the problem can thus be expected to be **polynomial in the best case and exponential in the worst case** as a function of the total number of disjunctions.

Hints on the Problem's Size The current data feature 22 tasks with 12 generic training means in 5 phases. The total number of unknown is however less than the product of these figures, since only part of the means can be available for a given task and in a given phase. In fact, there are 262 availability declarations for means to tasks in phases, ie. 262 utilization times to be determined. The number of constraints is of the order of 500. But the crucial point is not the number of continuous variables or the number of constraints. If the problem boiled down to solving the linear constraints, that would be easily achievable by Simplex-based tools. The number of disjunctions, or equivalently of 0-1 variables, is what really determines the complexity.

Let us consider only the maximum level-related disjunctions. With the current data sets, there are 63 disjunctive means. If, in order to minimize the overall cost eg., one (naively) enumerated all the combinations of utilization / non utilization of these means, and if for each combination a Simplex were run (with the idea of eventually computing the minimum of all linear minima) that would mean running the Simplex 2^{63} times, which is certainly not feasible in practice. If fixed familiarization times are added, all the means become 'disjunctive'. The size of the naive search space is then 2^{262} instead of 2^{63}, making the problem even more intractable. A realistic account of maturing would add another 88 disjunctions (4 phase level thresholds per task), which would give a figure of 2^{350}. It is clear that the entire search space need not be explicitly traversed. The figures are however huge and it is very unlikely that 'absolute' optima can be found within a reasonable time. A sensible objective is to provide good quality solutions within a proven distance to the theoretical optimum and in a relatively short time. This is still hard. For the time

being, the model will take into account **only the disjunctions originating from the maximum reachable level constraints**. If this can be solved in a proper way, further refinements will be introduced.

Previous Attempts to Solve the Problem The experts made several attempts to solve the problem, manually and by using spreadsheets. Optimization, indeed, can be achieved on each task separately by using a greedy algorithm, as far as no phase duration or global duration constraints are introduced. However, as soon as such constraint are stipulated, even for one task, no simple algorithm seems to allow to solve the problem. With several tasks, it is impossible to decide a priori how these constraints will 'distribute' over the different tasks, so that the problem cannot even be split. It can no longer be solved by manual or semi-automatic methods. This is what led to the idea of applying advanced resolution techniques.

Choice of CLP, ECL^{i}PS^e's Rational Solver and Parallelism

Choice of CLP For solving this problem, two major candidate techniques appear a priori: dedicated Mixed Integer-Linear Programming (MILP) packages on the one hand, and CLP on the other. A general discussion about CLP compared to Operations Research techniques in general can be found in [CFGG95a]. For this particular problem, the performance level to be expected from a MILP package is certainly higher than that of CLP, but the flexibility of CLP programming languages has been a strong argument in their favour. The requirement, indeed, is for a decision support system, and raw performance is not as important as the possibility for the user to interact with previous results (eg. introduce new pedagogical constraints). These results are possibly approximate (within a given distance to the optimum) but should be delivered in a short time. In addition, as far as performance is concerned, the number of continuous variables is not a limiting factor here. In contrast, ease of implementation of the heuristic methods is crucial, and for that purpose CLP languages have a decisive advantage over MILP packages. Finally, the system's specifications may evolve. CLP's declarativeness (meaning ease of expression and modification of the constraints and strategies) will much better allow the evolutions of the system. The *Artificial Intelligence and Advanced Computer Techniques Department* has a previous experience in the development of CLP-based decision support systems, mainly for production planning and scheduling [BCP92, CF94, CF95, BCF95] and has worked on methodological aspects [CFGG95a, CFGG95b]. User interaction principles designed for scheduling problems are also valid for TCO and will be applied.

Choice of a CLP Rational Solver It has been assumed so far that the problem would be handled by some combination of linear programming and enumeration methods. When choosing CLP languages like CHIP or ECL^{i}PS^e, one has the choice between linear solvers (LS) over continuous (rational) variables and also a Finite Domain (FD) solver. A comparison can be found in [CFGG95a, CFGG95b]. The Finite Domains are widely used for all kinds of planning and scheduling problems and need to be considered seriously. Below will be given arguments in favour or against the two solvers as far as TCO is concerned, and the conclusion reached.

In Favour of Finite Domains

TCO is a discrete problem, even for the utilization times, for in practice these times will be integer numbers (possible null) of learning hours or half-hours. It is not sensible to plan eg. that the pilot students should use a given mission simulator 2.37 hours in phase 2. One needs to round this figure, eg. to 2.5 hours.

The local propagation-based FD solver will handle larger-sized problems than the LS solver, which performs global manipulations over the constraint network. This is however probably not decisive here, since the number of constraints is not very large.

In Favour of a Linear Solver

Propagation is rather weak with the Finite Domains for the kind of disjunctive constraints encountered in TCO. Basically, one would use conditional propagation, viz. a disjunctive constraint would be frozen (leading to no propagation at all) until, as an effect of the decisions made by the program, one of its alternatives becomes either true or false for any possible values combination of its variables. This implies that a considerable amount of labelling would be required before some significant propagation is achieved. In addition, not the 0-1 variables need to be labelled (63 variables with the current data), as with a linear solver, but the utilization-time variables (262 variables with the current data), since propagation is not complete and only instantiation of all variables can ensure correct solutions. Global constraints have been developed in the Finite Domains for certain classes of problems, mainly in the area of scheduling, logistics and placement [BC94, CHI94, V93], but there is apparently no such global constraint that would fit the TCO planning problem. It is not a scheduling problem, not the problem of finding an order among training actions. The (now) classical disjunctive or cumulative constraints do not apply. TCO is a planning problem, where the issue is to decide at a high level which resources (the means) have to be used, which number of them are required and in which phases. The relatively weak handling of disjunctions achieved with conditional propagation might actually be acceptable if the aim were only to find a feasible solution. But one wants optimize, even if it is with some approximation, and the cost function is the sum of a large number of elementary costs. In the Finite Domains this often leads to a poor evaluation of the cost objective when the underlying propagation is weak (refer to the experience with the PSAP system in APPLAUSE). This was confirmed by experiments made at the beginning of the TCO project with the Finite Domain solver of CHIP.

In contrast, with linear solvers over continuous variables, linear relaxation techniques provide a global handling of the disjunctive constraints. A better evaluation of the cost objective may be expected, since the approximation of the not yet solved disjunctions combine linearly, whereas with Finite Domains an unsolved disjunction handled with conditional propagation would have simply no effect at all. For these reasons, the Linear Solver was preferred for TCO.

The rest of this document will be devoted to describing the constraints and strategies used in the prototype, and the way the system was parallelized.

Parallelism Applying OR-parallelism is a natural way of speeding up of the exploration of the choice points. More precisely, the initial expectations about parallelism for TCO were the following:

- in exploration phases, the users will want to get a set of significantly different acceptable solutions within a limited time; parallelism should help to increase the number of structurally distinct satisfactory solutions
- when the user has reached a stable definition of the problem, he may want to run the proof of optimality to completion (with some specified accuracy); here, parallelism is expected to substantially reduce the time needed for this proof.

4 Constraint Expression and Prototyping

The Mock-up The first mock-up was written in CHIP in 1993-1994, using the Rational Solver [B94]. It allowed to solve optimally only small problems (5 tasks). An algorithm for the distribution of the global constraints (such as maximum total duration) to the tasks was implemented, with which larger problems could be solved. But this approach was finally abandoned, since the distance of the thus obtained solutions to the actual optima could not be satisfactorily estimated.

A totally new mock-up was written at the end of 1994, which forms the base of the work in APPLAUSE. It was designed so as to give a pre-view of the final tool and, at the same time, facilitate the development work and the search for efficient resolution strategies. A high-level control of constraint setting and interaction with data is provided. The resolution core is implemented in CHIP / ECL^{i}PS^e, ie. it can run in both languages given a very limited number of specific predicate re-definitions. The mock-up has a graphical user interface for parameter acquisition and result display, written in CHIP's graphical layer (this has not been ported so far to ECL^{i}PS^e, since it can be used independently from the resolution to exploit results produced by the CHIP and ECL^{i}PS^e versions).

All examples of code given in this report will be in ECL^{i}PS^e.

Constraint Expression

Introduction of Linear Relaxations a) General Presentation

The disjunctions of the problem cannot be handled as mere choice points. As was already mentioned, the combinatorics are huge and it is simply impossible to construct all the alternative sets of linear constraints and for each of them run a linear optimization. The idea of linear relaxation is then to replace a disjunction of constraints by a conjunction, in a way such that

- the conjunction's solution set is a superset of the disjunction's
- it is as close as possible to it
- ways are provided to express the choices and when a particular branch is chosen, the solution set reduces dynamically to that of the branch.

This is achieved by introducing continuous variables for each choice point which are initially constrained to range between 0 and 1, and will eventually be assigned the value 0

or 1 (it is impossible to impose integrity constraints from the start, ie. to mix continuous and integer variables). These variables express the relative validity of the branches at any step of the computation. The best (tightest) possible linear relaxation of a problem consisting of linear constraints and disjunctions of linear constraints is the one whose solution polyhedron is the convex hull of the (non-convex) polyhedron defined by the initial problem. Computing it proves however very hard as soon as the initial problem features more than one disjunction. The intersection of the convex hulls of several disjunctions, indeed, is in general larger than the convex hull of their conjunction. Thus, computing the convex hull requires a time, and leads to a number of constraints, which may be exponential in the number of disjunctions in the worst case [DBH93]. Therefore, in general, looser relaxations are looked for, and several methods exist for this purpose. The basic relaxation method adopted for TCO is common in Mixed Integer-Linear Programming. Assume that the original disjunction is

$A_1.X \leq B_1 \vee A_2.X \leq B_2$

where A_1, A_2 are matrices with numeric coefficients, X is a vector of unknown, B_1, B_2 are vectors of numbers. The relaxation is the following:

$A_1.X \leq B_1\delta + U_1(1-\delta)$
$A_2.X \leq U_2\delta + B_2(1-\delta)$

where the vectors U_1 and U_2 are suitable upper bounds for $A_1.X$ and $A_2.X$, respectively.

This kind of relaxation may define the convex hull but may also be significantly larger [DBH93], depending on the dimension and the accuracy of the upper bounds. Reasonable bounds, however, can often be found from practical considerations related to the problem's semantics, which has been the case for TCO. It is still worth trying to tighten them. The closer the relaxation to the convex hull, the closer the cost of the linear optimum to that of the feasible optimum (ie. that of the best solution satisfying the integrity constraints on the δ variables), and the earlier useless branches will be eliminated in the search for the feasible optimum. As a matter of fact, if computing the convex hull were tractable, any search on the δ variables would be avoided — except some limited enumeration in certain degenerate cases — for the vertices of the convex hull (where linear optima are located) correspond to integer values of the δ coefficients. The experiments carried out for TCO to tighten the constraints by using knowledge about what they mean will be explained later.

b) Example

Consider the small example given in 3 a). The disjunctive constraint

$(L_3 \leq 50 \vee P_3 = 0)$

or, equivalently (since $P_3 \geq 0$)

$(L_3 \leq 50 \vee P_3 \leq 0)$

will be replaced by the relaxation

$L_3 \leq 50\delta + U_3(1-\delta)$,
$P_3 \leq V_3\delta + 0(1-\delta)$

where appropriate values for the upper bounds U_3 and V_3 need to be found. This is done by considering the maximum level that can be reached before m_3. It is that of m_2, 60, and this allows to take $U_3 = 60$. For V_3, it is easy to see that if the means m_3 is used, the progression cannot be more than its level, which is limited to 50, hence $V_3 = 50$. The relaxation is therefore (after eliminating the null term in the expression of P_3):

$L_3 \leq 50\delta + 60(1-\delta)$,
$P_3 \leq 50\delta$

Discrete Optimization Since at least some search is inevitable, this has to be conducted in such a way that useless steps are avoided: when a solution has been found with a given cost, it is no use searching for solutions that may have a higher cost. Branch & Bound methods have been developed for that purpose. For the TCO problem, procedures corresponding to *min_max* and *minimize* of the Finite Domains of ECL^{i}PS^e[ECL94] have been used. We will shortly recall the principles of these procedures. What relates to parallel execution will be explained in the section on parallelism.

a) Min_max

The principle is simple:

- usual depth-first backtracking search is performed;
- each time a new solution is found, the search is restarted from scratch, with the constraint that the cost should be strictly less than the cost of this solution;
- the procedure stops when the current search fails; then, either at least one solution has been found, in which case the last found one is optimal (the search for a better one has failed), or the problem has got no solution at all.

Practically, the stronger the constraints, the better the dynamic implicit evaluation of the cost lower bound performed by the constraint system and the quicker useless branches will be pruned.

A *min_max* for the rationals was first written from scratch for TCO, but finally it was found wiser to benefit from the built-in's safer implementation, and to use it with rational costs by simply rounding them down whenever a solution is found.

Finding approximate optima is often enough, as is the case for TCO. This is done by imposing a percentage: when a solution is found, then the search restarts with a new cost upper bound equal to the previous cost minus the percentage. Then, when the last search fails, the distance of the last solution (if there are solutions) to the absolute optimum is guaranteed to be less than the percentage. Time-limits can also be specified (this feature was implemented in ECL^{i}PS^e for TCO). Note that in that case if the system stops because of the time-limit no optimality (even by some percentage) has been proven.

b) Minimize

An alternative method consists in having the system backtrack with a new cost constraint when a new solution has been found, instead of re-starting the search from the top of the tree as with *min_max*. The idea is that no solution to the new constraint may have existed in the part of the search tree already visited, since no solution to the previous, weaker, constraint, existed there. All that was said above about the strength of the constraint system, the possibility of optimizing by n % or of imposing a time-out equally applies to that method. (It is, however, much harder to implement than *min_max*.)

Predicting which of the re-computation-based *min_max* or the backtracking-based *minimize* will perform better on a given problem is not easy, and one has to experiment. For TCO, both *min_max* and the minimize-like predicate with cost-parallelism were tried. The latter proved to be very efficient (this is explained in detail in 6).

c) Example

The equations of the small example given above easily translate to ECL^{i}PS^e code. Without *min_max*, enumeration of the branches leads to:

```
[eclipse 1]:
T1 $>= 0, P1 $>= 0, L1 $>= 0,
T2 $>= 0, P2 $>= 0, L2 $>= 0,
T3 $>= 0, P3 $>= 0, L3 $>= 0,
T4 $>= 0, P4 $>= 0, L4 $>= 0,

P1 $= 4*T1,  L1 $= P1,
P2 $= 8*T2,  L2 $= L1 + P2,
P3 $= 2*T3,  L3 $= L2 + P3,
P4 $= 10*T4, L4 $= L3 + P4,

L1 $<= 40,
L2 $<= 60,
D3 $>= 0, D3 $<= 1,
L3 $<= 50*D3 + 60*(1-D3),
P3 $<= 50*D3,
L4 $= 100,

Cost $>= 0,
Cost $= 1*T1 + 5*T2 + 2*T3 + 20*T4,

(D3 $= 0 ; D3 $= 1),    % ';' is the 'or' operator

rmin(Cost).             % linear minimization

P1 = 40
P2 = 20
P4 = 40
L1 = 40
L2 = 60
```

```
L3 = 60
P3 = 0
L4 = 100
T1 = 10
T2 = 2.5
T3 = 0
T4 = 4
D3 = 0
Cost = 102.5     More? (;)

P1 = 40
P2 = 10
P4 = 50
L1 = 40
L2 = 50
L3 = 50
P3 = 0
L4 = 100
T1 = 10
T2 = 1.25
T3 = 0
T4 = 5
D3 = 1
Cost = 116.25
yes.
[eclipse 2]:
```

With *r_min_max* (*min_max* adapted to rationals), the second branch is abandoned before its complete exploration, for it leads to a cost higher than the previous one:

```
[eclipse 3]:
T1 $>= 0, P1 $>= 0, L1 $>= 0,
T2 $>= 0, P2 $>= 0, L2 $>= 0,
T3 $>= 0, P3 $>= 0, L3 $>= 0,
T4 $>= 0, P4 $>= 0, L4 $>= 0,

P1 $= 4*T1,  L1 $= P1,
P2 $= 8*T2,  L2 $= L1 + P2,
P3 $= 2*T3,  L3 $= L2 + P3,
P4 $= 10*T4, L4 $= L3 + P4,

L1 $<= 40,
L2 $<= 60,
D3 $>= 0, D3 $<= 1,
L3 $<= 50*D3 + 60*(1-D3),
P3 $<= 50*D3,
L4 $= 100,
```

```
Cost $>= 0,
Cost $= 1*T1 + 5*T2 + 2*T3 + 20*T4,

r_min_max((D3 $= 0 ; D3 $= 1), Cost, 0), % optimum by 0% sought


r_min(Cost).

Current cost 102.5
Optimal cost 102.5

P1 = 40
P2 = 20
P4 = 40
L1 = 40
L2 = 60
L3 = 60
P3 = 0
L4 = 100
T1 = 10
T2 = 1.25
T3 = 0
T4 = 4
D3 = 0
Cost = 102.5
yes.
[eclipse 4]:
```

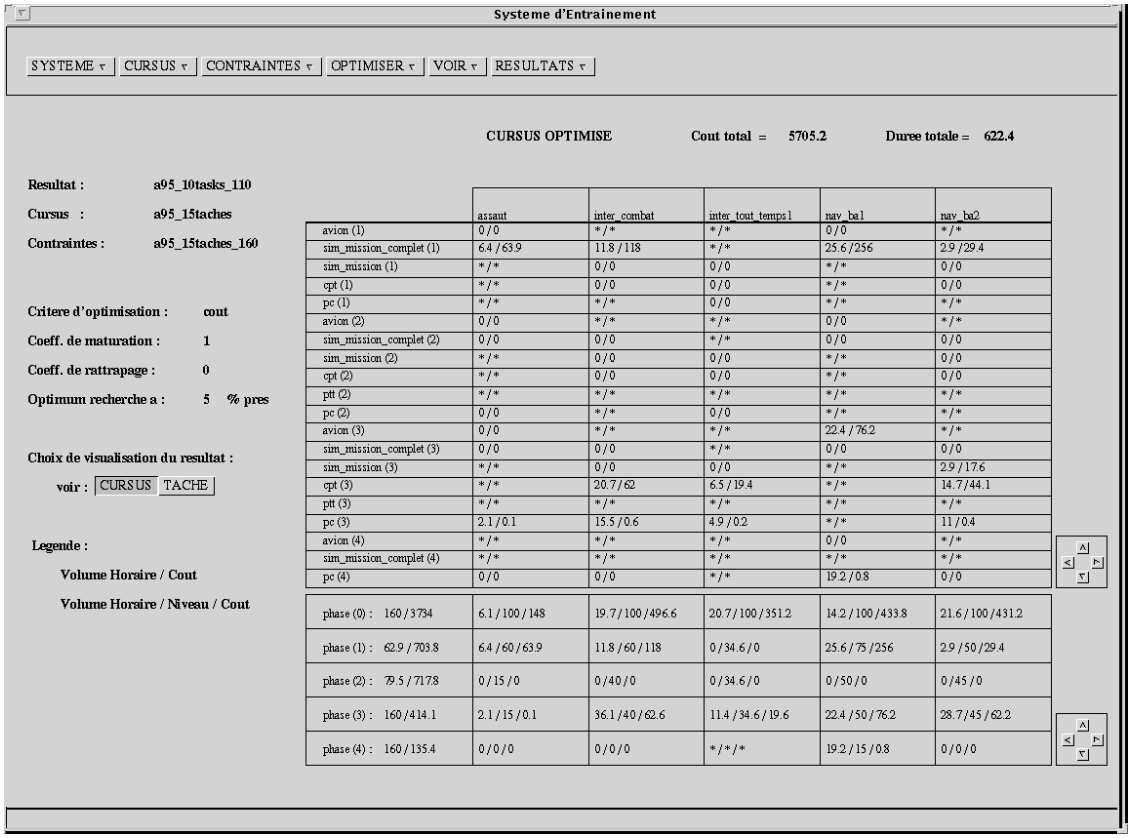
Linear minimization is performed after *r_min_max* in order to commit the cost to its actual minimum in the optimal branch. In this example, this is enough to instantiate all the problem variables. In general, an additional procedure is required for that, which has been omitted here for the sake of simplicity.

The TCO Program Operation The basic mechanisms have been explained. We will now say a few words on how the program works globally. For each resolution, the following operations are successively performed:

- The data and parameters are loaded and processed, and a single term representing the curriculum is constructed, whose characteristics will be accessed in the rest of the program in an object oriented-type way.
- The various types of constraints are successively set up. Failure may occur during this phase. In the operational system, information will then be displayed about when the constraint system has become inconsistent, and the user will be invited to reconsider at least some constraints of the latest introduced type.
- The discrete minimization procedure is run on the labelling of the decision variables, possibly with a time-limit and a specified accuracy. Experience has shown that

failure seldom occurs at this step, for most of the time solutions exist, even if at a very high cost. However, this is not guaranteed, and the operational system will have to provide practical hints for constraint relaxation.

- The optimum solution found is displayed in a spreadsheet form via the graphical interface (see Figure 4.1). A multi-solution variant of the resolution program based on the results of parallelism will be implemented and exploited at the user-interface level in the next version of the prototype.



	assaut	inter_combat	inter_tout_temps1	nav_ba1	nav_ba2
avion (1)	0 / 0	*/*	*/*	0 / 0	*/*
sim_mission_complet (1)	6.4 / 63.9	11.8 / 118	*/*	25.6 / 256	2.9 / 29.4
sim_mission (1)	*/*	0 / 0	0 / 0	*/*	0 / 0
cpt (1)	*/*	0 / 0	0 / 0	*/*	0 / 0
pc (1)	*/*	*/*	0 / 0	*/*	*/*
avion (2)	0 / 0	*/*	*/*	0 / 0	*/*
sim_mission_complet (2)	0 / 0	0 / 0	*/*	0 / 0	0 / 0
sim_mission (2)	*/*	0 / 0	0 / 0	*/*	0 / 0
cpt (2)	*/*	0 / 0	0 / 0	*/*	0 / 0
ptt (2)	*/*	*/*	*/*	*/*	*/*
pc (2)	0 / 0	*/*	0 / 0	*/*	*/*
avion (3)	0 / 0	*/*	*/*	22.4 / 76.2	*/*
sim_mission_complet (3)	0 / 0	0 / 0	*/*	0 / 0	0 / 0
sim_mission (3)	*/*	0 / 0	0 / 0	*/*	2.9 / 17.6
cpt (3)	*/*	20.7 / 62	6.5 / 19.4	*/*	14.7 / 44.1
ptt (3)	*/*	*/*	*/*	*/*	*/*
pc (3)	2.1 / 0.1	15.5 / 0.6	4.9 / 0.2	*/*	11 / 0.4
avion (4)	*/*	*/*	*/*	0 / 0	*/*
sim_mission_complet (4)	*/*	*/*	*/*	*/*	*/*
pc (4)	0 / 0	0 / 0	*/*	19.2 / 0.8	0 / 0

phase (0) : 160 / 3734	6.1 / 100 / 148	19.7 / 100 / 496.6	20.7 / 100 / 351.2	14.2 / 100 / 433.8	21.6 / 100 / 431.2
phase (1) : 62.9 / 703.8	6.4 / 60 / 63.9	11.8 / 60 / 118	0 / 34.6 / 0	25.6 / 75 / 256	2.9 / 50 / 29.4
phase (2) : 79.5 / 717.8	0 / 15 / 0	0 / 40 / 0	0 / 34.6 / 0	0 / 50 / 0	0 / 45 / 0
phase (3) : 160 / 414.1	2.1 / 15 / 0.1	36.1 / 40 / 62.6	11.4 / 34.6 / 19.6	22.4 / 50 / 76.2	28.7 / 45 / 62.2
phase (4) : 160 / 135.4	0 / 0 / 0	0 / 0 / 0	*/*/*	19.2 / 15 / 0.8	0 / 0 / 0

Figure 4.1: A Result Displayed by TCO

Performance of the Basic Program With the basic linear relaxation mechanisms that have been explained above and with no particular strategy for the labelling of the decision variables, performance is rather poor. Due to the combinatorial nature of the problem, an exponential behaviour in the worst case is inevitable, but it should be reduced to its minimum. Before parallelism was applied, several tracks were explored, some of which led to significant improvements both for sequential and, later, for parallel execution. They concern the improvement of the constraint system, on one side, and simple ordering principles for the decision variables inside the labelling routine, on the other side, and will be explained in section 6. We will first expose the principles of the parallelization of TCO.

5 Parallelization

This section is about the basic principles, initial expectations and first results regarding the parallelization of TCO. Optimization of parallelism will be dealt with in section 6, where a detailed account and representative figures are provided.

Parallelizing the Labelling Using OR-parallelism on choice points is the most natural idea, which means here parallelizing the assignment of 0 or 1 to the decision variables. It had to be checked that these choices are of enough coarse grain. This is, indeed, of utmost importance since the time spent in communication and reallocation of workers depends on it. Experience has proved that this parallelization is quite efficient, ie. of an acceptable grain. This is probably due to the fact that the most important decisions are put first (see section 6). The parallelizing strategy which has been used for the first experiments consists in using:

- an analog of *par_indomain* for rationals applied to the decision variables
- *min_max* as the minimization procedure.

The Program The program can be sketched as follows:

```
tco :-
        get_data(Data),
        set_up_constraints(Data, Decisions, Cost, UtilizationTimes),
        value_order(Domain01),              % [0, 1] or [1, 0]: parameter
        accuracy(Percent),                  % parameter
        r_min_max(label01(Decisions, Domain01), Cost, Percent),
        instantiate_all(UtilizationTimes). % ensure ground solution


label01([], _).
label01([D|Decisions], Domain01) :-
        r_par_indomain(D, Domain01),
        label01(Decisions, Domain01).


r_par_indomain(X, Domain) :-
        par_member(V, Domain),
        X $= V.
```

Results This strategy works well for the proof of optimality: the results show quasi-linear speed-ups with the hardest problem instances (see the figures with *min_max* in the tables given in section 6). They are, however, worse for the easier problems. Note that this parallelism has absolutely no effect on the time spent to find the solutions. This may seem disappointing, but probably only shows that the heuristic ordering of the variables

and the values (see 6) is good. Though these results were already satisfactory, it was felt that parallelism could bring more in terms of quality and variety of the solutions. This lead to a second phase of experiments, which is explained in section 6.

It is worth noting that this introduction of parallelism was extremely simple and required almost no specific debugging effort. This may be due to the fact that the system was initially programmed in a very declarative way, with no assert/retract eg., and with global variables used only in the pre-processing part. Only certain phenomena caused by the inherent asynchronism of parallel execution were slightly puzzling at the beginning. For example, if the results are written into a file in Prolog format, then a term and the following comma, or dot, written by a given worker should never be separated by a write statement performed by another worker in the same file; it is therefore crucial to use an atomic goal like

```
printf(Stream, "%q, ", [Term])
```

and not a complex goal goal like

```
write(Stream, Term), write(Stream, ", ")
```

Such features, however, are easily mastered.

The Rational Solver The ECL^{i}PS^e rational solver used for the experiments is clp(Q, R) [Hol95]. It is significantly slower than that of CHIP for example, which had several consequences:

- only small to middle-sized data could be tested within a reasonable time (between 9 and 21 decision variables, as compared to the 63 of the real-sized problem instances)
- it might be the case that with a faster solver the communication overhead ceases to be negligible and the speed-ups are slightly lower
- some strategies like having parallel choice points on dichotomic constraints for the phase durations could not really be tested, for the additional constraints resulted in unacceptable slow-downs.

6 Performance Debugging and Optimization

Improvements to Sequential (and Parallel) Performance In this section will be discussed the improvements made necessary by the poor performance level of the basic constraint model, and which proved valuable both for sequential and parallel execution. Enhancements relating specifically to parallelism are discussed in 6.

Improving the Constraint System a) Adding Constraints on the Levels to Separate the Branches

In the example already used, the relaxation

$L_3 \leq 50\delta + 60(1-\delta)$,
$P_3 \leq 50\delta$

expresses that either m_3 cannot be used ($\delta = 0$), or it may be used ($\delta = 1$) and the level L_3 reached after its (possibly null) utilization is less than its ceiling (50). However, nothing does actually prevent L_3 to be less than 50 with $\delta = 0$, and P_3 to be zero with $\delta = 1$. In other words, the configuration $L_3 \leq 50,\ P_3 = 0$ is common to both branches, ie. a disjunction has been expressed, but not a mutual exclusion. This is logically not a problem, but procedurally may lead a computational redundancy and slow down performance.

To obtain separated branches, one needs to express eg. that

- either the level reached before m_3 is not greater than its ceiling, and m_3 may be used
- or this level is greater than the ceiling of m_3, and m_3 may not be used.

This means that the original disjunction has to be re-written as

$L_2 \leq 60$,
$((L_2 \leq 50, L_3 \leq 50) \vee (L_2 > 50, P_3 = 0))$

where $L_2 \leq 50$ and $L_2 > 50$ guarantee that no solution to the global problem common to the two branches can be found. Since $L_2 \leq 50$ is implied by $L_3 \leq 50$, the disjunction can be simplified, giving

$L_2 \leq 60$,
$(L_3 \leq 50 \vee (L_2 > 50, P_3 = 0))$

The relaxation is straightforward:

$L_2 > 50(1-\delta)$,
$L_3 \leq 50\delta + 60(1-\delta)$,
$P_3 \leq 50\delta$

Since strict inequalities are not always handled efficiently by linear solvers, an arbitrary small quantity ϵ can be introduced:

$L_2 \geq 50(1+\epsilon)(1-\delta)$,
$L_3 \leq 50\delta + 60(1-\delta)$,
$P_3 \leq 50\delta$

Though one constraint is added to each disjunction, performance has been improved by a factor between 2 and 7 on the data sets that have been tried, due to the removal of

redundant computations. The method exposed in the next paragraph, however, subsumes this improvement.

b) Adding Propagation Constraints Between Decision Variables

If a disjunctive means in a given phase cannot be used because its maximum level is overstepped by the utilization of some other means before it, then no means of the same or of a subsequent phase having a ceiling lower or equal to its own can be used either, for their maximum levels will be also overstepped. In terms of decision variables, this means that if the one attached to the given means is assigned value 0, the variable of these other means has also to be 0. This is only partially expressed by the equations written so far. Let us consider the example below with an additional means $m_{3'}$, belonging to the same phase as m_3 and also disjunctive but with a higher ceiling (55). We obtain the following system:

$L_2 > 50(1 - \delta_3)$,
$L_3 \leq 50\delta_3 + 60(1 - \delta_3)$,
$P_3 \leq 50\delta_3$,
$L_2 > 55(1 - \delta_{3'})$,
$L_{3'} \leq 55\delta_{3'} + 60(1 - \delta_{3'})$,
$P_{3'} \leq 55\delta_{3'}$

If $\delta_{3'}$ takes value 0, δ_3 should as well. In practice, when $\delta_{3'} = 0$ occurs, then $L_2 \geq 55$ is imposed, but this implies only that δ_3 should be at least equal to 0.5 in order that $L_3 \leq 50\delta_3 + 60(1 - \delta_3)$ be satisfied. The way of enforcing value 0 to be propagated in such a case is quite simple: it is enough to add a new constraint between the decision variables:

$\delta_3 \leq \delta_{3'}$

to ensure that $\delta_3 = 0$ will be entailed by $\delta_{3'} = 0$, or $\delta_{3'} = 1$ from $\delta_3 = 1$. When added systematically to the previous equations these new propagation constraints lead to a slight performance improvement. More interesting is the fact that when the branch-separating constraints ($L_2 > 50(1 - \delta_3)$ and $L_2 > 55(1 - \delta_{3'})$ are removed, then performance is improved by a factor of 10. This may mean that the main effect of the branch-separating constraints was in fact to achieve some propagation between the decision variables, which is now taken over in a more efficient way by the new constraints, and that the branch-separating action in itself was not so crucial. But this explanation has not been proven. Anyway, in practice, the best system for this example appears to be

$L_3 \leq 50\delta_3 + 60(1 - \delta_3)$,
$P_3 \leq 50\delta_3$,
$L_{3'} \leq 55\delta_{3'} + 60(1 - \delta_{3'})$,
$P_{3'} \leq 55\delta_{3'}$
$\delta_3 \leq \delta_{3'}$

even though it may allow some redundancy between the branches. (It might be the case that some of the separation constraints could be kept, but this has not been yet investigated.)

In general **a $\delta_j \leq \delta_i$ constraint has to be written between any two means m_i and m_j such that the maximum level of m_j is lower or equal to that of m_i and m_j is in the same phase as m_j or in a subsequent one**. There may be in principle some redundancy between these constraints due to transitivity. To eliminate this, a simple ad hoc procedure using the Finite Domain solver has been written for TCO, which computes the minimal set of propagation constraints to be set up. In practice, however, redundancy is seldom encountered and this procedure may be skipped.

Improving the Labelling a) Ordering the Decisions

The reason for ordering the decisions is to be able to make first those decisions which have the most significant impact on the most constrained aspects, ie. here cost and durations. Which decisions have more influence was hard to determine a priori, and experiments were necessary. The results are simple and they are explained below.

Ordering the Decisions for each Task

A number of tests have been performed on the order of the decisions for a given task. The basic choice is whether to start from the first phases (the beginning of the training) or the last ones. Starting from the end, ie. ordering the variables in antichronological order of the phases and decreasing order of means' ceilings inside each phase, seems to be statistically better, in particular for the proof of optimality, which is about 1.5 faster that the reverse order on the tested examples. It is likely that this order enforces stronger constraints on the curriculum. These results, however, are in no way absolute, and could be reconsidered. The ordering is simply an execution parameter. Attempts to start eg. with the means with highest cost or highest maximum level have not proved conclusive.

Ordering the Tasks

With the current labelling, the decisions for a task are completely made before the system goes to the next task, and performance is extremely sensitive to the order in which the tasks are handled. Attempts to correlate this to simple criteria like the number of disjunctive means in the task, the means' costs, etc, have first failed. Finally, a good agreement between execution time and a synthetic criterion was found. This criterion is the *absolute efficiency of the two-seater combat aircraft* (coefficient 'k') for the task. Starting with the tasks with the lowest values of k pays off most of the time. The meaning of k is the following. The efficiencies given in the data are relative values, expressed by convention as a percentage of the efficiency of the two-seater combat aircraft (100 %). If a means is declared as having a relative efficiency of 50 % eg., learning with it is assumed to be twice slower than with the two-seater. This is conventional since the two-seater may not be adapted at all for learning the task in the particular phase (beginners cannot use it for instance). Important is the overall consistency of the relative values: if a means in a phase has an efficiency of 30 % and another one of 60 %, then learning with the second one should be twice as fast. Computing k for a given task is done by reference to the current curriculum, assuming that it allows to reach level 100%. The normalization formula is the following:

$$k * \sum_i RelativeEfficiencies_i * CurrentUtilizationTime_i = 100$$

As a consequence, k^{-1} is a measure of the (virtual) time that would be required to reach level 100 if only the two-seater were used. The tasks with lowest k are therefore the most demanding, those which are likely to be the most expensive and/or the longest ones, and it is sensible that making decisions first on them improves performance. The improvement is indeed very significant compared to a random order of the tasks, since execution time is divided by a factor ranging between 10 and 50 (the greatest difference is observed with the tests that take the longest times). In practice, tests have become handleable in a few minutes in CHIP that would not be run to completion beforehand.

Ordering the decisions first by phases, then by tasks (eg. by increasing k), though expected to give good results, has not proved interesting.

b) Ordering the Values

Ordering the decisions is not enough. The order in which values 0 and 1 will be tried has to be fixed. This can be done for each decision independently, either statically (ie. once and for all before the labelling), or dynamically, just before the decision is made. It may consist eg. in computing the lower bound for the cost by linear optimization with both candidate values and taking as first choice the one giving the lower cost. The experiments carried out statically did not prove convincing, probably because the assessment of the cost is not enough accurate before the labelling starts. Doing it dynamically on the other hand is extremely expensive. Starting with either 0 or 1 for all the tasks is much simpler and gives better results. The experiments have shown that starting with 0 is in general better, for statistically the optimum curriculum are obtained by using few disjunctive means. (Stochastic methods for value ordering have also been tried in sequential search; they will not be explained here — see [Per94]).

Improvements to Parallel Performance

Improving the Quality and Variety of the Solutions As was mentioned above, the first and most natural parallelizing strategy using *min_max* and *par_member* proved to be already a satisfactory utilization of parallel resources, yielding quasi-linear speed-ups for the hard problem instances. The lack of improvement for the first solutions is not a problem, since these are anyhow obtained rapidly with the ordering heuristics described in 6. One might, however, expect to get better quality solutions, and a greater number of them. This could be achieved to some extent by running *min_max* with with a low percentage (ie. a high accuracy), but this would be detrimental to the length of the proof of optimality, which on hard problem instances changes of order of magnitude between eg. 10% and 5%, or 5% and 1%.

The solution was provided by *Cost-Parallelism* (CP) in Branch & Bound [PM94, PM95]. The idea behind CP is that a part of the parallel resources can be usefully applied to exploring various cost bounds, instead of having all workers exploiting the parallel annotations of the program. With CP, in addition to the *conservative search* (the one performed with the usual *minimize* or *min_max*), *optimistic searches* and/or a *pessimistic search* are also performed. Assume that at some time during the resolution process, there is a global cost bound G, used by all the searches. The pessimistic search has then $Cost < G$ as cost constraint, the (possible multiple) optimistic searches use $Cost \leq E * G - F$ (where E is

the accuracy as a fraction of 1, and F is a fixed integer which is different for each optimistic search), and the conservative search uses $Cost < E * G$ in the same way as with the usual *minimize* or *min_max*. Whenever a solution is found by one of the searches, all of them are given a new, tighter bound. Whenever a search fails, all searches with a tighter cost constraint are abandoned and their workers reallocated. It is expected from the optimistic searches that they will go quickly to very good solutions by cutting short several cost steps and that the loss of one or more workers for the parallel annotations of the program will be compensated by the speed-up of the search for good solutions. The task of the pessimistic search in contrast is rather to shorten the proof of optimality by gradually improving the solution. This tightens progressively the cost bound, which in general makes the proof easier. It has also the effect of yielding better solutions for a given required accuracy than would have been obtained otherwise. Let us illustrate this with an example. With a 10% accuracy ($E = 0.9$), if a solution has just been found with cost 1000, the conservative search will proceed with a cost upper bound of 900 (ie. what an ordinary *min_max* or *minimize* would do). The pessimistic one will simply look for solutions stricly better than 1000. An optimistic search may look for better than 800, assuming a $F = 100$. If now the pessimistic worker finds the next solution, eg. with cost 950, then the conservative search will proceed with a cost constraint of 855 (10% better than 950), the pessimistic one will look merely for stricly better than 950 and the optimistic one with 755 (ie. 855 minus the fixed 100). Each time a solution is found by a conservative or optimistic search, it is likely to be further improved by the pessimistic search. And each time a solution is found by the pessimistic worker, the proof of optimality with the specified percentage is made easier, since the cost constraints get tighter. Optimality is proven when the conservative search fails (or the pessimistic search, which is rather unlikely for it is less constrained). The experiments carried out at ECRC have shown that pessimistic search will usually pay off with sufficiently high optimization percentages.

CP is implemented in two predicates: *cp_min_max* uses a restarting strategy and has only conservative and optimistic searches; *cp_min* uses a backtracking strategy (it is a minimize-like predicate) and has conservative, optimistic and pessimistic searches. The *cp_min* predicate has been adapted to rationals for TCO by ECRC. The sketch of the TCO program using it is exactly the same as with *min_max* (5). Very little adaptation has been necessary; this amounts practically to adding a cost-monitor argument in the labelling procedure. After various experiments, the combination of pessimistic and optimistic search has been found the most effective and has been extensively tested in comparison with *min_max*. The results are illustrated by the tables given below. They are very satisfactory. With *cp_min(opt, pess)* and a 5% or 10% accuracy

- the whole search including the proof of optimality according to the specified accuracy is completed for most of the tests with *significant speed-ups*, quasi-linear (sometimes even super-linear) in the number of workers for the hardest instances
- *a number of solutions are obtained, quite diverse in their structure, some of them being in fact very close to the absolute optimum* (this can be seen from runs with 1% accuracy).

We will now explain and comment in detail several tests that constitute consistent and complete sequences.

The Tests

The example data sets are subsets of the real data. They have been chosen so as to feature a sufficient number of disjunctions, but not too many linear equations. This has been achieved by taking the hardest piloting tasks, in terms of number of disjunctive means. Data *t04* features the four most difficult tasks, with a total of 21 disjunctions (ie. a third of the disjunctions of the full problem), with no phase or curriculum duration constraints. Data *t04.50* is more realistic: the tasks are the same but maximum duration constraints for all phases have been added. Smaller data have been also tested, consisting of two tasks, with 9 disjunctions (*taa1sr.25.20.10*) and 12 disjunctions (*taa2sr.25.20* and *taa3sr.30.20*). These smaller data have all relatively tight phase duration constraints.

All tests have been performed on SunSparc20 work stations with 4 processors. For each test the results are given for

- sequential execution; parallel execution with 2, 3, 4 workers
- *min_max* and *cp_min* in parallel execution; in sequential mode only *min_max*, since cost-parallelism does not make sense
- optimization by 10%, 5%, 1% (except for *t04* and *t04.50* which take too much time with 1%).

The tests have been executed several times (with one exception). The times measured in sequential mode are all very similar for the same parameters and the number and quality of solutions are always identical. Therefore, the results for sequential runs are simply

- the number of solutions found
- the cost of the optimum solution (rounded)
- the time taken to find this optimum (geometric mean over the different runs)
- the time taken for the whole search including the proof of optimality (geometric mean).

For the parallel runs the number and quality of the solutions may vary from one run to the other. To reflect this diversity, for each test (given data, optimization predicate, accuracy, number of workers), the following results are provided:

- the list of the numbers of solutions produced during the search, for all runs
- the different optimal costs obtained, with the indication of the type of search which led to them if the optimization predicate is *cp_min* (*c* for conservative; *o* for optimistic; *p* for pessimistic)
- the geometric mean over all runs of the speed-ups for the computation of the optimum solution, with respect to the sequential execution
- the corresponding standard deviation; geometric mean and standard deviation are computed according to [Pre94a], based on [Ert94]; notice that a standard deviation of 1 means no deviation (all results identical)

- the geometric mean of the speed-ups of the search including the proof of optimality
- the standard deviation.

No figure has been given for the proof of optimality alone. The reason is that this hardly makes sense with *cp_min*, since in many cases this proof actually starts before the optimum is found and its bound is gradually refined as better solutions are produced by the pessimistic search. The time when it actually starts does not seem accessible. So, only the total time and the associated speed-up are meaningful. Note that if the time for the proof of optimality was to be defined as the time elapsed between the moment when the optimum is found and the end of the whole search, then many of the speed-ups would be higher than those given in the tables. It often takes, indeed, a longer time to the pessimistic search to produce the optimum than it does in sequential search (but the optimum is better).

In order not to overload the tables, the times are not given for the parallel tests, the speed-ups being more significant. (To compute the mean parallel times, simply divide the sequential times by the speed-ups.)

taa1sr.25.20.10		10%		5%		1%	
		min_max	cp_min	min_max	cp_min	min_max	cp_min
Seq.	Nbr of solutions	1	-	2	-	6	-
	Optimum	933	-	871	-	861	-
	Time optimum	5.1	-	22.7	-	75	-
	Time optimality	42.7	-	60.0	-	135	-
2 Wkrs	Nbr of solutions	1,1,1,1, 1,1,1	6,6,6,6, 6,6,6	2,2,2,2, 2,2,2	6,6,6,10, 6,6,6	6,6,6,6 6	15,15,14, 14,14
	Optimum (wkr)	933	893(p)	872	861(c), 867(c)	861	855(p)
	Speedup optimum	0.98	0.27	0.99	1.18	0.8	1.01
	Std deviation	1.02	1.02	1.03	1.23	1.02	1.03
	S-up optimality	1.39	1.75	1.19	1.72	0.95	1.65
	Std deviation	1.04	1.02	1.02	1.03	1.01	1.01
3 Wkrs	Nbr of solutions	1,1,1,1, 1,1,1	5,5,5,5, 5,5,5	2,2,2,2, 2,2,2	7,9,7,7, 9,7,9	5,5,4,4, 5	15,15,15, 16,17
	Optimum (wkr)	924, 933	908(p)	866, 871	861(c), 888(c)	855, 861	861(p)
	Speedup optimum	0.95	0.43	1.4	1.14	1.1	1.33
	Std deviation	1.03	1.03	1.06	1.09	1.06	1.04
	S-up optimality	1.52	2.26	1.44	1.96	1.2	1.87
	Std deviation	1.02	1.02	1.05	1.1	1.04	1.03
4 Wkrs	Nbr of solutions	1,1,1,1, 1,1,1	5,5,5,5, 5,5,5	2,2,2,2, 2,2,2	8,7,7,8, 7,8,7	5,5,5,5, 6	14,19,17, 15,13
	Optimum (wkr)	924, 933	908(p)	866, 871	861(c), 893(c)	855, 861	855(p), 861(p), 862(p)
	Speedup optimum	0.93	0.41	1.43	1.17	0.99	1.38
	Std deviation	1.03	1.04	1.07	1.05	1.04	1.12
	S-up optimality	1.53	2.64	1.46	2.51	1.13	2.09
	Std deviation	1.02	1.03	1.03	1.05	1.03	1.06

Comments on taa1sr.25.20.10

This is an easy problem instance, perhaps too easy, since increasing the number of workers to more than 2 does not pay off much, neither in terms of speed-up nor in terms of quantity of solutions produced during the search and quality of optimum. *cp_min* is better than *min_max*: it is most of the time faster for the whole search, it always gives more solutions during the search, it provides better optima in most cases. When compared to sequential search, parallelism normally gives better optima, but there are exceptions (eg. here 888 in *cp_min* 5%). These are cases when a parallel annotation of the program has led to an optimal solution quicker than the left-most path followed in sequential mode, but this solution, though optimal according to the required accuracy, is worse than the sequential one.

Note, and this is a general remark, that the speed-ups on the time taken to find the optimum, are not really significant. In the cases when the first sequential solution is already optimal according to the required accuracy (which means that the heuristics worked well),

then if this solution is also the first one found in parallel, the speed-up will be about 1 (slightly less due to the communications overheads), otherwise *anything* can be observed: worse or better optimal solution, speed-up > 1 (as with these data) or < 1 (see the next tests).

All these tables show only the cost as a characteristic of the various solutions. These solutions actually have a great *strucural diversity*, even if the costs are similar. They correspond to significantly different utilizations of the training means. This diversity has two sources:

- Exploitation of the parallelism of the program is the first source. Very high in the search tree, an alternative to the left-most path is explored in parallel and leads to a good solution. Since the decisions with greatest impact on the curriculum are put first, this solution is significantly different.
- When the cost bound gets close to the global optima, producing new solutions implies backtracking rather high in the search tree, which has the same effect.

This structural diversity is present in all the tests.

taa2sr.25.20		10%		5%		1%	
		min_max	cp_min	min_max	cp_min	min_max	cp_min
Seq.	Nbr of solutions	1	-	1	-	3	-
	Optimum	893	-	893	-	861	-
	Time optimum	7.4	-	7.5	-	41.0	-
	Time optimality	146.7	-	165.5	-	221.1	-
2 Wkrs	Nbr of solutions	1,1,1,1, 1,1,1	4,4,4,4, 4,4,4	1,1,1,1, 1,1,1	4,4,4,4, 4,4,4	3,3,3,3,3	6,6,5,5 6
	Optimum (wkr)	893	861(p)	893	861(p)	861	861(c)
	Speedup optimum	0.98	0.51	0.97	0.52	0.93	2.0
	Std deviation	1.01	1.02	1.04	1.02	1.02	1.01
	S-up optimality	1.71	2.49	1.7	1.57	1.35	1.15
	Std deviation	1.02	1.02	1.02	1.02	1.01	1.01
3 Wkrs	Nbr of solutions	1,1,1,1, 1,1,1	5,5,5,5, 5,5,5	1,1,1,1, 1,1,1	5,5,5,5, 5,5,5	3,3,3,3,3	6,7,7,6, 6
	Optimum (wkr)	893	861(p)	893	861(p)	861	861(c)
	Speedup optimum	0.97	0.49	0.97	0.5	0.8	1.65
	Std deviation	1.02	1.03	1.02	1.02	1.04	1.04
	S-up optimality	2.1	3.35	2.19	2.51	1.42	1.37
	Std deviation	1.02	1.1	1.02	1.02	1.02	1.02
4 Wkrs	Nbr of solutions	1,1,1,1, 1,1,2	6,6,6,6, 6,6,6	1,1,1,1, 1,1,1	6,6,6,6, 6,6,6	3,3,3,3,3,	7,7,6,5, 5
	Optimum (wkr)	893, 885	861(p)	893	861(p)	861	861(c)
	Speedup optimum	0.94	0.48	0.95	0.48	0.7	1.5
	Std deviation	1.02	1.03	1.03	1.03	1.03	1.05
	S-up optimality	2.37	4.18	2.54	3.05	1.43	1.47
	Std deviation	1.03	1.03	1.02	1.03	1.02	1.03

Comments on taa2sr.25.20

Notice here that parallelism does not pay off with 1% and that with this accuracy *cp_min* is not better than *min_max*. Generally speaking, pessimistic search is not expected to do well with low percentages like 1%, for the pessimistic and conservative searches will have almost the same bound. But with 10% eg. the results can be very good, as is the case here. They are satisfactory also with 5%. It is worth noting that the pessimistic search with 10% and 5% has given solutions that are in fact optimal by 1% (and possibly less)! In practice, of course, one will not normally attempt to prove this optimality by 1%, and only 10% or 5% will ge guaranteed. But it is satisfactory to know that quite often the solution given is actually much better than the required accuracy.

taa3sr.30.20		10%		5%		1%	
		min_max	cp_min	min_max	cp_min	min_max	cp_min
Seq.	Nbr of solutions	1	-	2	-	4	-
	Optimum	999	-	944	-	944	-
	Time optimum	10.1	-	51.1	-	82.9	-
	Time optimality	53.5	-	89.5	-	192.4	-
2 Wkrs	Nbr of solutions	1,1,1,1, 1,1,1	5,5,5,5, 5,5,5	1,1,1,1, 1,1,1	5,5,5,5, 5,5,5	3,4,4,5,3	7,7,7,7 6
	Optimum (wkr)	980	950(p)	980	950(p)	944	944(p)
	Speedup optimum	1.11	0.6	5.59	3.27	0.96	1.42
	Std deviation	1.03	1.02	1.02	1.03	1.13	1.03
	S-up optimality	1.26	1.74	1.27	1.89	1.18	1.34
	Std deviation	1.0	1.02	1.02	1.02	1.07	1.03
3 Wkrs	Nbr of solutions	1,1,1,1, 1,1,1	6,6,6,6, 6,6,6	1,1,1,1, 1,1,1	6,6,6,6, 6,6,6	4,3,3,4,4	8,7,7,8, 7
	Optimum (wkr)	980	950(p)	980	950(p)	944	944(p)
	Speedup optimum	1.12	0.63	5.68	3.02	0.98	1.15
	Std deviation	1.04	1.02	1.05	1.09	1.16	1.04
	S-up optimality	1.31	2.05	1.51	2.31	1.26	1.51
	Std deviation	1.02	1.02	1.02	1.04	1.1	1.02
4 Wkrs	Nbr of solutions	1,1,1,1, 1,1,2	5,7,7,7, 7,6,7	1,1,1,1, 1,1,1	7,7,7,6, 6,7,7	3,3,3,3,2	7,8,7,7, 8
	Optimum (wkr)	980	950(p)	980, 955	950(p)	944	944(p)
	Speedup optimum	1.11	0.6	5.7	3.16	1.27	1.03
	Std deviation	1.04	1.08	1.03	1.03	1.09	1.02
	S-up optimality	1.34	2.0	1.72	2.92	1.5	1.78
	Std deviation	1.02	1.14	1.15	1.03	1.05	1.02

Comments on taa3sr.30.20

cp_min is significantly better here than *min_max* in all respects. Parallelism pays off with 10% and 5% (the speed-ups are less than linear, but still significant). Note that the parallel runs by 5% lead to solutions that are worse than that of the sequential search, a phenomenon already observed on *taa1sr.25.20.10*. It is interesting to explain in detail what happened in the sequential executions and the ones with eg. 2 workers:

- Sequentially, the first solution produced is 999. It is not optimal by 5%. The next one has cost 944 and is optimal.
- With *min_max* and 2 workers, the exploitation of a parallel annotation leads quickly to an optimal solution with cost 980. But the proof of optimality is slow (total speed-up only 1.27), for 980 is a bound significantly less stringent than 944.
- With *cp_min* the unique worker of the conservative search (the other worker being used by the pessimistic search) finds the sequential solution with cost 999. Then the pessimistic search achieves a progressive refinement of the optimal cost (5 solutions are produced). As a consequence, the cost constraint of the conservative search is gradually strengthened and this results in a greater speed-up for the whole search than with *min_max* (speed-up 1.89).

t04		10%		5%	
		min_max	cp_min	min_max	cp_min
Seq.	Nbr of solutions	1	-	1	-
	Optimum	1437	-	1437	-
	Time optimum	7.2	-	7.3	-
	Time optimality	612.8	-	14803.5	-
2 Wkrs	Nbr of solutions	1,1	7,7	1	7,7
	Optimum (wkr)	1437	1414(p)	1437	1414(p)
	Speedup optimum	0.83	0.06	1.04	0.06
	Std deviation	1.27	1.21	1.0	1.2
	S-up optimality	1.73	1.76	2.06	1.49
	Std deviation	1.19	1.19	1.0	1.01
3 Wkrs	Nbr of solutions	1,1	8,8	1,1	8,8
	Optimum (wkr)	1437	1414(p)	1437	1414(p)
	Speedup optimum	0.89	0.05	0.92	0.06
	Std deviation	1.12	1.13	1.12	1.16
	S-up optimality	2.43	3.04	2.56	2.5
	Std deviation	1.15	1.15	1.2	1.17
4 Wkrs	Nbr of solutions	1,1	9,9	1,1	9,9
	Optimum (wkr)	1437	1414(p)	1437	1414(p)
	Speedup optimum	0.76	0.05	0.74	0.05
	Std deviation	1.15	1.18	1.28	1.25
	S-up optimality	2.82	3.62	3.1	3.16
	Std deviation	1.23	1.21	1.25	1.29

Comments on t04

This test and the next one are harder. The speed-ups of the total search time with *cp_min* are quasi-linear with 10%, a little less with 5% but still good. The 1414 solution is actually optimal by less than 0.1% (this was shown by separate tests, not shown here). And, as usual, several solutions are produced during the search (all of them in this case being already optimal by the required accuracy). Note once again that the speed-ups on the optimum are meaningless, for the optimum eventually found is not the same as in sequential mode.

t04.50		10%		5%	
		min_max	cp_min	min_max	cp_min
Seq.	Nbr of solutions	1	-	1	-
	Optimum	1475	-	1475	-
	Time optimum	27.3	-	27.6	-
	Time optimality	1306.7	-	14865.9	-
2 Wkrs	Nbr of solutions	1,2	5,5	1,1	5,5
	Optimum (wkr)	1475, 1470	1440(p)	1475	1440(p)
	Speedup optimum	0.9	0.28	0.95	0.27
	Std deviation	1.31	1.24	1.26	1.27
	S-up optimality	1.86	6.98	1.84	1.13
	Std deviation	1.15	1.24	1.22	1.25
3 Wkrs	Nbr of solutions	1,1	6,6	1,1	6,6
	Optimum (wkr)	1470	1440(p)	1470	1440(p)
	Speedup optimum	1.26	0.24	1.31	0.24
	Std deviation	1.29	1.35	1.25	1.35
	S-up optimality	2.68	8.03	2.55	2.02
	Std deviation	1.31	1.23	1.33	1.33
4 Wkrs	Nbr of solutions	1,1	7,7	2,1	7,6
	Optimum (wkr)	1470	1440(p)	1470	1440(p)
	Speedup optimum	1.49	0.31	1.05	0.24
	Std deviation	1.03	1.02	1.45	1.3
	S-up optimality	4.04	11.25	3.07	2.6
	Std deviation	1.04	1.03	1.39	1.47

Comments on t04.50

The interesting point here is 10% with *cp_min* which shows *super-linear* speed-ups of the whole search.

General Conclusions on the Tests

cp_min with optimistic and pessimistic search, together with parallel annotations in the labelling procedure, is a very good parallelization strategy when optima are sought with a 5% or 10% accuracy. It is better than *min_max* with parallel annotations. The benefits, compared to sequential executions, are:

- better solutions for a given required accuracy, often close to the absolute optimum; this is not guaranteed, but it seems to be a general tendency
- much more solutions produced during the search, structurally different, a number of them being actually optimal by the required accuracy
- a speed-up of the whole search which is of the order of magnitude of the number of workers.

From the TCO user's point of view, the structural diversity of a set of solutions close to the optimums is extremely important. There is not *one* solution to the curriculum

optimization problem, but several, with possibly similar financial costs but quite different pedagogical and organizational implications. The user has to be given as many of these solutions as possible and it is up to him to choose the most appropriate one.

Other Possible Tests A number of potentially interesting experiments could not be carried out before the end of APPLAUSE:

- *minimize* could be tried instead of *min_max* (see 4). This would require a specific adaptation of the predicate to the rationals, as was done for *cp_min*.
- A time-out was implemented for *min_max*. Once it is available also for *cp_min*, one could measure the number and quality of the solutions generated within a given time-limit. It is already clear from our experiments that *cp_min* bears a definite advantage over *min_max*, but this could be quantified.
- Once the optimum cost has been assessed, by 5% eg., one could search for all solutions not worse than eg. 5% of the optimum found (which would mean solutions not worse than roughly 10% of the absolute optimum) and measure the speed-up as a function of the number of workers. This is no longer minimization but a *findall*-type search, and quasi-linear speed-ups are expected.
- Other hardware platforms will have to be tried, with more processors. This will allow a better exploitation of the parallelism of the program (with only 4 processors, one is used for the pessimistic search, one for the optimistic, and only two remain for the parallel annotations, which is very little).

7 Conclusion

In sequential execution CLP had proved adequate in the search for good solutions, since efficient (though simple) heuristics could be designed. Parallel execution, with cost-parallelism and pessimistic search, has made it possible to produce in a shorter time a much greater number of good quality solutions of quite different structures. When the proof of optimality is required, this is achieved with a speed-up of the total search which is of the order of magnitude of the number of workers. The difficulty encountered in the parallelization process proper have been very small and the time spent for parallelization is significantly less than for each of the other phases: specification, modelling, expression of the constraints, sequential optimization. The main conclusion is that parallel resources, when available, can be efficiently and easily exploited in ECL^{i}PS^{e}for this kind of application.

It is interesting to note that the results are rather predictable, not for individual runs of course, but as to the general tendencies. The fact that *cp_min* with pessimistic search would pay off for the proof of optimality and the quality of the solutions had been indeed predicted by the ECRC team before the tests were actually performed. It seems that some confidence can be attached to a few general rules. A useful methodological advance would probably consist in presenting these rules (with 'confidence intervals') in a systematic way and making them accessible to the average Parallel CLP programmer.

TCO has induced the development in the ECL^{i}PS^{e}system of several general purpose features, like time-out and adaptation of the minimization predicates to rationals. We hope that at a more general level the work on this application has contributed to the understanding of the potential benefits of parallel CLP.

8 Acknowledgments

Jean-Marie Saget, as an acknowledged expert of the domain, laid the foundations of the study on pilot training optimization. He provided the basic model of the problem and the data. Together with Jacques Louis and Grégoire Veber, he welcomed our work with a constant interest and support. The developement of the sequential CLP system is a common work of Annie Fischler and the author of this report. Denis Béja and Grégory Perrat did hard work on this problem when they were at Dassault. Thanks to the APPLAUSE partners for the very fruitful and friendly atmosphere in the project, to the ECL^{i}PS^{e}team at ECRC in particular for their highly qualified help.

Chapter 5.

A Decision Support System for the Venice Lagoon

Giuditta Festa, Giuseppe Sardu and Roberto Felici

1 Problem description

The main role of EDS-Systems & Management in the APPLAUSE project was that of investigating the suitability of the ECL^{i}PS^e environment for the development of Decision Support Systems. The Decision Support field represents a commercially important class of applications which is meeting remarkable interest and is subject of research efforts. In particular, our application addresses a social and economical critical aspect: environmental monitoring and control. The availability of information on the natural environment is increasing much more rapidly than the development of tools for the interpretation and the use of such information. Furthermore, more and more crucial is becoming the need for systems which will support effective management of the ecosystem. Particular features that must be supported include the timely availability of data on the dynamic changes in the ecosystem, the ability of the system to provide decision support tailored to a particular legislative framework and also the ability to provide forecast of the outcome of particular decisions taken by the managing authority. We have planned the Venice Lagoon DSS, trying to meet these requirements. In the following we shall outline the DSS development and the experiences we have gathered working in the ECL^{i}PS^e environment.

1.1 The Venice Lagoon and its safeguard

The unique character of Venice mainly derives from its tight relation with the Lagoon on which it dominates. Alas, the Lagoon, though conferring a charming look to the city, is afflicted by the growing environmental pollution effects.

The ecosystem of the Venice Lagoon is a complex system involving a number of different elements (cities, rivers, tides,...) and a wide territory which even includes regions distant from the Lagoon but having significant effects on it, because of the action of rivers or other diffusion means. An index of the complexity of such an ecosystem is the great extension of the geographical area which affects the Lagoon environment; it is composed by:

- the proper Lagoon territory, whose borders can be demarcated by the Brenta mouth on the south and the town Iesolo on the north;
- the whole hydrological basin connected to the Lagoon. It concerns a wide area crossed by rivers, channels but also underground streams which will flow in the Lagoon;
- the range of sea bordering the Lagoon. The water coming from the sea, through the tides, interacts with the Lagoon environment by activating exchange and dilution mechanisms.

Trying to categorize the different pollution sources, three main classes can be identified: agricultural, urban and industrial. The agricultural pollution is mainly due to the introduction in the Lagoon of polluting substances by rivers coming from other regions intensively cultivated. As for the urban pollution, it mainly consists of organic pollution arising from the presence of inhabited areas, hospitals, tourist accommodations; it is noteworthy that Venice doesn't have sewers nor it is possible to build them. The industrial pollution originates from the discharges of the industrial plants. The Lagoon is

home to some of the most important chemical and oil-refinery sites of northern Italy. The effects arising from the discharges of such a relevant quantity of polluting substances in the Lagoon are of great danger for the balance of the whole ecosystem but even for the inhabitants' health and the state of preservation of the Venetian historic buildings. It is, therefore, easy to understand the relevance of the Lagoon safeguarding problem.

The necessity to efficiently carry out a controlling activity affecting the levels of pollution in the Lagoon was already felt during the bygone centuries. To this end, the Venice Water Magistracy was founded in 1501. Its institution originated from the need to acquire an intimate knowledge of the issues connected with the whole Lagoon environment, to ensure timely and efficient actions aimed at safeguarding the Lagoon and its inhabitants. The Water Magistracy was in charge of the centralized management of all matters which might have had effect on the Lagoon hydraulic system: the state of the channels which cross the city, the control of the discharges in the Lagoon. Over the centuries, the duties of the Water Magistracy have not been basically modified, but the conditions affecting the area constituting the environment of the Lagoon changed substantially. Suffice it to mention the growth of the population, the creation of the industrial site of Porto Marghera, the changeover, for both nature and quantities, of the substances discharged in the Lagoon. Nowadays the Venice Water Magistracy has to keep control on thousands of discharges, besides facing the analysis of an increasing number of factors, which the most recent studies in the field of anti-pollution have turned out to be of great interest.

The activity of the Venice Water Magistracy is carried out in compliance with a number of regional, national and European laws and directives. Such regulations state the inspections that have to be performed, but also prescribe threshold values to which pollution parameters must conform. One of the most important tasks which is in charge of the Water Magistracy is the management of the granting of licenses for new discharges. As each single new emission of polluting substances must obtain a specific authorization, the natural or legal persons who are involved with the emissions are requested to supply the Water Magistracy with data broadly regarding:

- the quantity and the nature of the discharged substances;
- the treatment plants which will be performed to reduce the polluting effects of the discharged substances.

The Water Magistracy staff have to check that the substances discharged from the new emission point do not cause a threshold value violation. For this they have to compare the declared values with the ones indicated by the different laws and directives which regulate this scope. Of course, they are also in charge of the checks related to the authenticity of the declared data; this involves an activity of water sample collection and analysis. Against detected irregularity the Water Magistracy can force the culprit of the offending emissions to introduce more effective treatments and, in very critical situations, it can even revoke the license.

Since recent, this time-consuming administrative activity has been supported by an automated system which deals with the whole bureaucratic procedure relative to the granting of the emission licenses. This system essentially consists in a database storing information (emitted substances and declared quantities, location of the polluting source, depuration treatments, status of the file in the bureaucratic procedure and so on) relative to

each discharge for which a license has been requested. Objective of such an operational environment is that of responding to the pollution effects instead of acting to prevent contamination, and as such it does not allow to evaluate the effects of a new discharge on the Lagoon ecosystem. Even the granting of a new emission license is, at present, only subordinated to the observance of some static threshold values. Such values are stated by laws and are relative to the maximum allowed concentration values of substances in the emissions, but don't take into account the actual pollution in the location where the substances are to be discharged. A further refinement for supporting the Water Magistracy decision making activity may concern an optimization phase, as in fact, besides obtaining a complete overview of the pollution state, this institution is interested in finding the best solution to face critical situations.

Considering the requirements suggested by the Venice Water Magistracy experts we have envisaged a number of problems which seemed suitable to be tackled with the methodologies put at our disposal by ECL^{i}PS^e. We intended to develop a decision support tool whose aim is not in totally substituting the human responsible and expertness but rather in improving its ability by a thorough analysis of the possible technical solutions. This has been the origin of the Venice Lagoon DSS; we will now describe the DSS functionalities and its implementation.

1.2 A DSS for the Venice Lagoon

Two different phases can be identified in the decision making process: a first *data interpretation phase* and a further *decision making phase*. In our context, the data interpretation phase consists in getting the pollution status of the Lagoon and comparing it with an acceptable one, in order to point out alarming discrepancies. The decision-making phase consists in planning technical interventions aimed at restoring an acceptable state at "as low a cost" as possible. To cover such two phases the Venice Lagoon DSS is composed by:

- a database containing environmental data;
- a hydrodynamic model of the Lagoon;
- a knowledge-based core;
- an interface module for the end-user.

Its structure together with the interactions is depicted in the figure 1.1.

Here is a brief description of the DSS components and of their relationship.

The environmental database

The environmental DB provides information on polluting sources and polluting substances which are necessary for the evaluation of the pollution states in the Lagoon. The input data for the DSS is described by a set of predicates that supply it with the knowledge on the physical state of the Lagoon, the bounds imposed by government regulations, and the

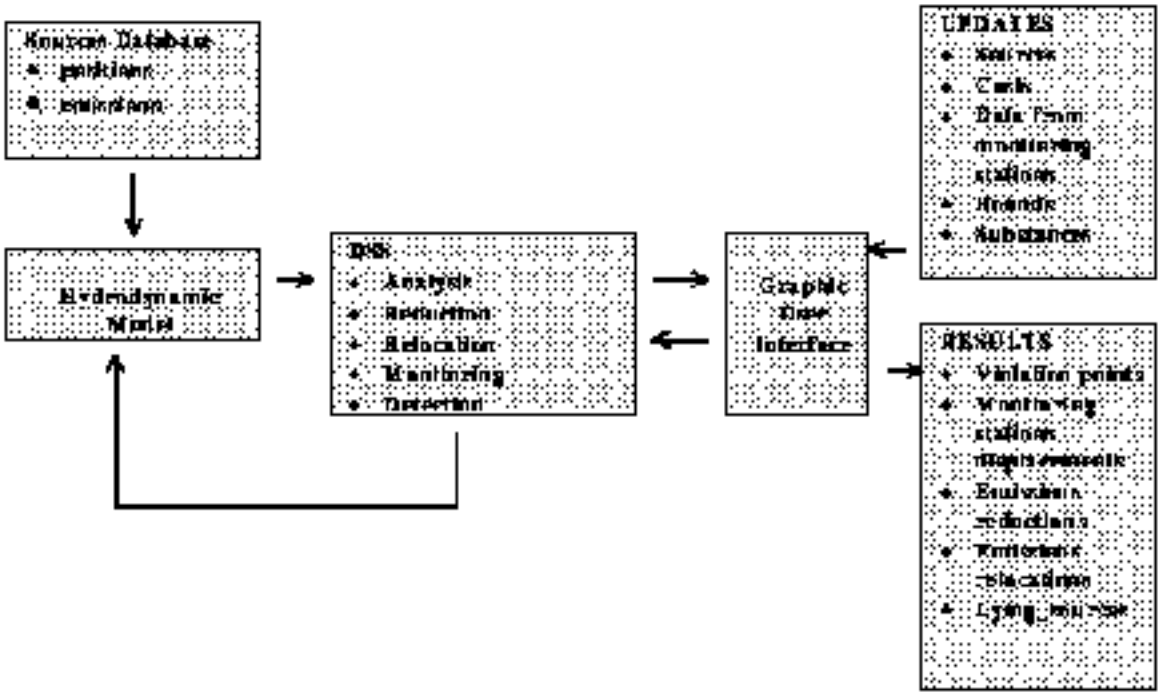


Figure 1.1: The components of the Venice Lagoon DSS

cost parameters necessary for the optimization problems. The most important predicates are listed in the following:

`source(Id_No, X, Y)` which describes the positions of the pollution emitting sources;

`substances(Id_No, Name)` describing each substance name;

`emission(Source, Substance, Quantity)` indicating the Quantity of Substance emitted by Source;

`e_bound(Substance, Bound)` stating the maximum concentration value allowed for Substance in the Lagoon;

`cost_redc(Source, Substance, List_of_Costs)` indicating the cost necessary to reduce the quantity of Substance emitted by Source;

`cost_relc(Active_source, Inactive_source, Cost)` indicating the cost necessary to relocate the emission of the `Active_source` on the `Inactive_source`;

`concentrationFactor(Source, X, Y, Factor)` representing the factors which, multiplied by the quantity of substance emitted by `Source`, return the expected concentration value for the substance in point `(X,Y)`.

The DSS prototype we have developed for the APPLAUSE project considers 12 different polluting substances and up to 80 polluting sources.

Unlike the objective nature of the data relative to the quantities emitted by the sources and the allowed bounds, the costs are dependent from subjective evaluations. In this context, in fact, costs do not represent "simple" monetary values, rather they denote a more

complex evaluation of social, economical and technical factors whose "value assessment" can only arise from the experience gathered daily "on the field" by the staff which is involved with the Lagoon safeguard.

The hydrodynamic model of the Lagoon

The role of an hydrodynamic model in such a DSS consists in describing the Lagoon hydrodynamic behaviour to simulate the diffusion of polluting substances and, therefore, to forecast their concentration values in the Lagoon.

The implementation of a mathematical model, tailored for a complex ecosystem such as the Lagoon environment and designed to meet a high degree of accuracy, should have required remarkable efforts; so, for the current phase of prototyping, we have deemed to be sufficient to build a simplified model, which anyway could generate the necessary inputs to the interpretative knowledge-based system.

Such basic hydrodynamic model accepts as input the following data:

- a polygon broadly describing the Lagoon morphology;
- a list describing the positions of the three Lagoon access points to the sea;
- a list describing the positions of the polluting sources in the Lagoon.

Considering the physical equations relative to the diffusion of substances in a liquid and a number of simplified hypothesis (no viscosity, two dimensional movements,...), the model produces as output a matrix, whose coordinates represent physical lagoon coordinates and cells contain lists of *Source, Factor* couples indicating the emitting sources and the concentration factor relative to the emitting source *Source* in point `(X,Y)`. This matrix is then converted into the set of

```
concentrationFactor(Source, X, Y, Factor)
```

statements required as input by the DSS.

Due to the large amount of calculations necessary, we have implemented the hydrodynamic model using the C language. We have decided to use ASCII files to implement the communication between the model and the DSS optimization core written in ECL^{i}PS^e. This means that the model reads an ASCII file to obtain its input data and, in turn, puts the resulting concentration factors in another ASCII file, which will then be read to produce the ECL^{i}PS^e `concentrationFactor/4` predicates. We have opted for such a solution to make the model as much as possible independent from the DSS optimization core; in such a way, the simplified model can be easily replaced by a more sophisticated one as soon as it becomes available.

As the concentration factors depend on the position of the sources, it is necessary to re-run the hydrodynamic model to obtain new concentration factors each time the number or the location of the polluting sources change. This can be done directly from the DSS.

The knowledge-based core

The logic core of our DSS is composed of several conceptual modules, corresponding to different problems and phases which can be activated independently. As said, they have been conceived to meet the requirements of the DSS potential end user: the Venice Water Magistracy. The modules consist essentially of constrained search and optimization tasks, so the use of ECL^{i}PS^e for their realization has allowed to exploit the main features of this language concerning parallel programming and constraint handling. The tasks covered by the modules are explained in the following:

Analysis: It calculates the concentration values for the polluting substances in the Lagoon, compares such values with the maximum bounds that are imposed by law, and gives as result a list of points where the bounds are exceeded.

Reduction: It aims at restoring the pollution back to legal values in the locations in which the previous Analysis has discovered violations. To obtain such a result, the most cost effective reduction plan for the emissions of the polluting sources is suggested.

Relocation: The relocation strategy constitutes an alternative solution to the problem of the excessive concentration levels. To reduce the concentration values in the points detected by Analysis, the emissions are not reduced, but they are partially relocated in some other admissible areas (inactive_sources). The problem aims at minimizing the global cost arising from the relocation of the emissions.

Monitoring: This task aims at planning a monitoring network. It finds a plan for dislocating the minimum number of monitoring stations which allow to have each polluting source controlled by a monitoring station.

Detection: It compares the concentration values collected by a monitoring campaign with the ones predicted by the hydrodynamic model. As a result it returns a list of polluting sources which may emit more than they have declared.

The user-interface

A considerable part of our development efforts has been devoted to supply the DSS with an efficient and user friendly interface. The **user interface** constitutes an important feature for a DSS whose target user is not necessarily skilled in the computer science field. We have used graphical displays as much as possible to give the user an immediate and intuitive grasp of the situation of the Lagoon. All the tasks performed by the DSS are made accessible by a menu-bar which, in turn, shows tiled-menus. On the screen a map of the Venice Lagoon is always visible; the results of the requested tasks are superimposed over it. An on-line help has been conceived to support the user in the DSS navigation; it supplies brief explanations of the tasks included into the system and detect sequences of not allowed operations and wrong inputs introduced by the user. Furthermore, the user-interface, allows to graphically update the data (costs, emissions, polluting source location, ...) which define the Lagoon environment.

As for the user-interface implementation, we have used a public domain graphic tool: Tcl/Tk. Tcl and Tk are two software packages providing a programming system for easily developing and using graphical user interfaces. In particular, Tk is an extension to TCL allowing to construct Motif-like user interfaces. The connection between Tcl/Tk and ECL^{i}PS^e is supported by Pro_Tcl, a Prolog interface to the Tcl/Tk toolkit, which has been built by ECRC. Tcl/Tk commands are made accessible from ECL^{i}PS^e through the `tcl_eval/2` predicate (which accepts any Tcl expression and passes it to the Tcl interpreter; Pro_Tcl also allows to call ECL^{i}PS^e predicates from a Tcl command or script and to get back the value of variables. Pro_Tcl frees the ECL^{i}PS^e user from the burden of plunging into the interconnection issues, in fact it does the job in a user transparent way.

2 Characterization

As explained in the previous section, the Venice Lagoon DSS is an experimental application whose realization has started within the APPLAUSE project. The problems composing the DSS have been studied and jointly pointed out with the Venice Water Magistracy staff. Even though such problems arise from the daily controlling activity, and therefore refer to existing needs, no previous attempts to scientifically solve them had been made. Our first investigation concerned in particular the feasibility of an ECL^{i}PS^e - based approach to the handling of the tasks that our DSS was intended to cover. The question was: *"Is (Parallel) CLP adequate for our application field?"*.

From a general viewpoint, an ecosystem is a dynamic system, intrinsically non-linear, in which every decision can either have no consequence, by virtue of the homeostatic power of the system, or induce dangerous degeneration because of the priming of progressive concatenations of damages to the whole system. On the other hand, part of the decision process (e.g. minimizing costs of planned reductions of polluting emissions) can be viewed as the activity of solving *optimization problems*. In particular, the Water Magistracy decision-making activity, even though relying on quantitative estimations, is based on logical definitions of casual relations, laws, regulations and technical knowledge; such further characteristic advised us to investigate other techniques than the traditional mathematical ones (e. g. Operations Research) having a consolidated history in the problem solving world. An interpretative model, therefore, inspired by mathematical logic rather than by classical analysis, seemed to best support the knowledge processing and qualitative reasoning. Actually, in the context of our application, the constraint logic programming paradigm has been adopted to bridge the gap between knowledge processing and optimization problems.

The problems we had to tackle were characterized, among other things, by a large amount of computation required to generate, compare and combine the large amount of hypotheses representing all the possible solutions to reduce the pollution levels. Constraints allowed the knowledge representation and execution, but only a parallel exploration of such large search spaces could obtain the performances levels necessary to achieve reasonable execution times.

The initial set of problems (mainly Reduction and Relocation) constituting the DSS were, therefore, characterized by:

- a non-linear nature
- a large search space (just to give an order of magnitude: the search space of Reduction for 80 sources is 11^{80}!)

and the joint action of constraints and parallelism put at our disposal by the ECL^{i}PS^e environment seemed to be a valid candidate to efficiently tackle them.

During the project development, we have worked to achieve a correct and efficient modelling for our problems in terms of variables and constraints. To this regard, it must be pointed out that the declarative nature of ECL^{i}PS^e has made the introduction of additional constraints in the programs an easy task. In some cases (e. g. for the Analysis problem), we got aware of the fact that the nature of the problem (Analysis is essentially an unconstrained search problem on a large data set) discouraged the use of a logic programming approach and we have exploited the possibility, given by ECL^{i}PS^e, to use an imperative language ("C") to give an alternative representation of it.

Unlike Reduction and Relocation, the other two problems (Monitoring and Detection) don't require the joint action of constraints and parallelism; for both of them, the set of constraints which defines the problem succeeds in heavily pruning the search space so that even the sequential version works with low execution times. It must be pointed that the use of constraints has made the representation of such two problems simple and intuitive.

3 Constraint Modelling and Prototyping

In this section we shall describe, in a more detailed way, the problems that constitute the logic core of the Venice Lagoon DSS. During the DSS development, our main efforts have been spent in finding the most adequate constraint modelling to represent the different modules. The selection of constraints able to prune the search space as much as possible has turned out to be fundamental to obtain acceptable execution times for problems with a large search space, as are the ones composing our DSS.

As already stated, our optimization problems are characterized by non-linear relations and high complexity. The Propagation over integers with Finite Domains, provided by ECL^{i}PS^e, has allowed to efficiently face the non-linearity and the combinatorial nature of the problems. Where a linear representation of the problem was possible, we have tested the ECL^{i}PS^e linear solver over rationals, which uses the Simplex algorithm. The conclusion we can draw on this comparison, which of course only refers to linear or linearizable problems, are the following: when the number of constraints is too low to ensure a valid pruning of the search space, the Simplex method turns out to be far more efficient than the enumerative method; whereas, when the set of constraints is meaningful enough, the two methods have comparable performances.

3.1 Constraint modelling for the DSS modules

The modelling of a problem is always an exacting job. We have introduced constraints in a progressive manner, stating initially only the most obvious ones necessary to supply

a preliminary formalization of the problem, and then trying to adopt more specific and complex constraints, suitable for a more efficient and complete pruning of the search space.

In the following, each module composing the DSS will be described in terms of variables and constraints.

The Analysis problem

The Analysis module covers the *data interpretation phase* of the decision making process. It aims at detecting the points in the Lagoon where the concentration value for a given substance exceeds the maximum allowed bound.

In a point (X, Y) the concentration value for a substance Sub is given by:

$$Conc(Sub, X, Y) = f_{s1}(X, Y) * e_{s1,Sub} + \cdots + f_{sN}(X, Y) * e_{sN,Sub}$$

where:

$f_{si}(X, Y)$ is the concentration factor for the source si in the point (X, Y) (given by the `concentrationFactor(Source, X, Y, Factor)` fact in the database).

$e_{si,Sub}$ is the quantity of the substance Sub emitted by the source si (given by the `emission(Source, Substance, Quantity)` fact in the database)

$s1, \ldots, sN$ are the active sources

Our first modelling of the problem consisted in collecting (via the `findall/3` predicate) the (X, Y) points for which the disequation

$$Conc(Sub, X, Y) \geq Bound(Sub)$$

was verified. Such points were then recorded in a Prolog fact

```
violations(Sub, [(X1, Y1, Gap1), ..., (Xn, Yn, Gapn)].
```

The resulting execution times were unacceptable (for 80 sources we obtained execution times of 6600 sec.!).

A further and more thorough study about the nature of the Analysis problem has definitely shown that a logic programming approach was not the best candidate to tackle this problem. In fact, Analysis is essentially an unconstrained search problem on a large data set, whose goal is to calculate the pollution concentration values for a relevant number of points in the lagoon. Analysis spends most the time accessing the `concentrationFactor/4` predicates, which represent the concentration factor for a point and an emitting source in the Lagoon. To reduce such access times we have decided to exploit the possibility of using the "C" language to imitate the action of a Prolog predicate. So we have used the C

language to insert the concentration factors in a C table in order to make them accessible using the source number and the coordinates of the point as indexes.

The introduction of these C procedures has not compelled us to rewrite the Analysis program, we have just had to declare the predicate `concentrationFactor/4` as *external* in order to link it to the correspondent C function (`p_get_factor`); in such a way, a reference to the `concentrationFactor/4` predicate in the Analysis program results in a call to the `p_get_factor` function whose code has been previously compiled and loaded.

Using this access strategy we have reduced the execution time by a factor of 20.

The Reduction problem

The Reduction procedure computes a reduction plan for the sources emissions in order to bring the concentration values relative to the violation points back to the allowed ones. The violation points are those detected by a previous running of the Analysis procedure.

For a given substance Sub and for each violation point (X, Y, Gap) listed in the fact

```
violations(Sub, [(X1, Y1, Gap1), ..., (Xn, Yn, Gapn)].
```

the constraint stated is:

$$f_{s1}(X,Y) * e_{s1,Sub} * V_{s1} + \cdots + f_{sN}(X,Y) * e_{sN,Sub} * V_{sN} \geq Gap(X,Y)$$

where

$f_{si}(X,Y)$ is the concentration factor for the source si in the point (X,Y)

$e_{si,Sub}$ is the quantity of the substance Sub emitted by the source si

$s1, \ldots, sn$ are the active sources

V_{si} is a domain variable which represents the percentage of necessary reduction relative to the source si. The domain for such variables is composed of the integers in the interval between 0 and 10. This means that the emissions can be reduced by 0%, 10%, 20%..... 100%.

Reduction is an optimization problem and it aims at minimizing the following cost function:

$$C_{s1,Sub,Vs1} + C_{s2,Sub,Vs2} + \cdots + C_{sN,Sub,VsN}$$

being $C_{si,Sub,Vsi}$ the cost to reduce the emission of the source si by the percentage indicated by the variables V_{si}. Such costs are given by facts as:

```
cost_redc(Source, Substance, [C0, C10, C20, ..., C100]).
```

The parameters for the Reduction problem are:

- the number of active sources
- the number of violation points
- the domain cardinality for V_{si} variables.

The dimension of the search space is given by:

$$domain\ card.\ for\ V_{si}\ variables^{number\ of\ active\ sources}$$

and can become really large. Just to give an idea: with 11 values in the domain (from 0 to 10) and 80 sources, it becomes 11^{80}.

Reduction is therefore a non-linear minimization problem with a large search space, so it is particularly suited to be tackled by CLP. The non-linear nature of the problem is inherited by the non-linearity of the reduction costs. According to the Water Magistracy experts' opinion, non linear costs may best represent the combination of economical, social and technical factors which are affected by a reduction of the emissions.

The resulting execution times are highly sensitive to data in the sense that some sets of violation points (and therefore of constraints) cause a strong pruning of the search space, whereas in some other cases the pruning action is not sufficient and the consequent exploration of the search space by enumeration results in a time-consuming activity (more than 3 hours for 80 sources).

Of course, we have aimed at reducing the execution times for the worst case.

Our attempts to find further constraints for a stronger pruning of the search space has not be completely successful, so we have tried to reduce the execution times by other means.

In a first version of the Reduction procedure we stated a constraint for each violation point detected by Analysis. Here is a set of execution times we have obtained:

	20 sources	40 sources	80 sources
Violations	cpu time		
50 - 180	2.1 sec	5.98 sec	+ 2h
100 - 500	2.5 sec	15.17 sec	+ 3h
380 - 800	5.9 sec	46 sec	+ 3h

Such results were, in a certain sense, unexpected. In fact one could think that with a higher number of violation points (and, consequently, a higher number of constraints) lower execution times would have resulted as a consequence of a stronger pruning of the search space. As a matter of fact, a logic explanation exists for the rise in the execution times: if a subset of violations points are contiguous, they have very similar concentration values, and therefore, the resulting constraints do not cause further pruning of the search space, but only increase the computational efforts required for the propagation method on a large number of constraints. Following this observation, we have grouped the violation

points by locality; this means that, if a subset of violation points are contiguous, we state the constraint only on the one amongst them having the highest concentration value.

A further action to obtain acceptable execution times has consisted in making a smaller search space by reducing the cardinality of the V_{si} domain variables (it represents the base for the dimension of the search space). We have considered 6 values in the domain (instead of 11) obtaining a remarkable reduction for the execution times (60 sec. vs. 2h for 80 sources).

The Relocation problem

Like Reduction, the Relocation procedure aims at reducing the concentration values in the violation points pointed out by a previous running of the Analysis procedure. To obtain such a result, it devises a transfer plan of emissions from some active sources to other inactive ones.

With the term "inactive source" we indicate potential places where polluting substances can be discharged without causing critical levels of pollution. Obviously those areas should be characterized not only by current low pollution levels. Firstly, they must be "law eligible", meaning that they should not, explicitly or implicitly, be "emission restricted" areas, as an example mussels cultivation areas (hopefully!) low pollution levels, but it is forbidden to locate a polluting source in their premises. Moreover, there are certain technical and socio-economical considerations which must be taken into the due account, a shallow water area, is not, from the technical viewpoint, a candidate as an alternative discharge place. Similarly, there are places either too close to critical areas, monuments or historical palaces, or interested by water streams flowing towards critical areas which are not obviously admissible. So, as one can easily understand, inactive sources selection is a very complex task requiring a deep knowledge of several Lagoon aspects, and as such is conducted on *ad hoc* basis by the Water Magistracy staff.

For a given substance Sub and for a violation point (X, Y, Gap) listed in the fact

```
violations(Sub, [(X1, Y1, Gap1), ..., (Xn, Yn, Gapn)].
```

the constraint stated is:

$$F_{s1,Is1}(X,Y) * e_{s1,Sub} * Perc_{s1} + \cdots + F_{sN,IsN}(X,Y) * e_{sN,Sub} * Perc_{sN} > Gap(X,Y)$$

where:

$F_{si,Isi}(X,Y) = f_{si}(X,Y) - f_{Isi}(X,Y)$ is the difference between two concentration factors and can be seen as representing the convenience in transferring a percentage of the emissions from the active source si to the inactive source indicated by the variable I_{si}

$e_{si,Sub}$ is the quantity of the substance Sub emitted by the source si

$s1, ..., sN$ are the active sources

$Perc_{si}$ is a domain variable which represents the percentage of emission relocation for the source si. The domain for such variables is between 0 and 10. This means that the emissions can be relocated by 0%, 10%, 20%..... 100%.

I_{si} is a domain variable which indicates which is the inactive source to which part of the emissions of the active source si must be relocated. The domain for such variables is the set of the inactive sources.

Relocation is an optimization problem and it aims at minimizing the following cost function:

$$Perc_{s1} * C_{s1,Is1} + \cdots + Perc_{sN} * C_{sN,IsN}$$

being $C_{si,Isi}$ the linear cost to relocate the 10% of the emission of the source si to the inactive source indicated by the variable I_{si}. Such costs are given by facts as:

```
cost_relc(Active_Source, Inactive_Source, Cost).
```

A further kind of constraint is imposed for each inactive source. It states that the sum of the emissions relocated to an inactive source must be lower than the allowed value ($Bound$):

$$\sum_{i=1}^{N} e_{si,Sub} * Perc_{si} \leq Bound_{Sub}$$

The parameters for the Relocation problem are:

- the number of active sources
- the number of inactive sources (it is the domain cardinality for I_{si} variables)
- the number of violation points
- the domain cardinality for $Perc_{si}$ variables.

The dimension of the search space is given by:

$$(num\ of\ inact.\ sources * domain\ card.\ for\ Perc_{si}\ vars)^{num\ of\ act.sources}$$

considering 11 values for the domain, 15 active sources and 2 inactive sources the cardinality for the search space becomes $(2 * 11)^{15}$!

The non-linear nature of the Relocation problem appears in the structure of the constraints which define it. Each constraint involves two kinds of domain variables: the first one ($Perc_{si}$) indicating the percentage of emission that must be relocated, the second one (I_{si}) representing the inactive source where the relocation must be routed. The choice of

considering only linear costs for the relocation problem arises from the need of avoiding a further increase in the problem complexity.

No further constraints have been pointed out for a stronger pruning of the search space, so we have worked on the above described ones to reduce the execution times as much as possible. Such times have turned out to be very high even for a low number of active sources and a small set of violations:

	20 sources	40 sources
Inactive sources	cpu time	
2	26.1 sec	1741 sec
3	559 sec	+3 h
4	6052 sec	+3 h

The presence of non-linear terms in Relocation has made necessary the introduction of a user-defined constraint. This type of constraint forces the propagation even where the ECL^{i}PS^e automatic propagation over linear terms cannot not deduce any new information. We shall relate about it in the section 5.2.

In order to obtain lower execution times we have tried:

- to reduce the search space by considering a smaller domain for the $Perc_{si}$ variables. The resulting computation times are shown in the below:

	11 domain values	6 domain values	4 domain values
Inactive sources	cpu time		
2	26 sec	20 sec	9 sec
3	559 sec	190 sec	20 sec
4	6052 sec	5400 sec	104 sec

- to introduce pre-computation rules aiming at detecting, in a preliminary phase, the violations which can't be solved by the relocation strategy. Such rules will be addressed in section 5.1.

The Monitoring problem

The Monitoring problem devises a dislocation plan for the monitoring stations in order to obtain information about the effective concentration of polluting substances in the Lagoon.

A monitoring station is a tool having its own technical range; this means that a source is controlled by a monitoring station if it is located within the technical range of the station. Our aim was to minimize the number of monitoring stations (whose cost is relevant) by placing them in the potentially most dangerous points in the Lagoon. Since concentration factors in a point P decays with an exponential law which is function of the distance from the polluting sources, we can consider the polluting sources as the most dangerous points. According to this consideration we have only considered the positions of the polluting sources as potential locations for the monitoring stations.

Monitoring aims at minimizing the number of necessary monitoring stations, in a way such that *each source* will be within a radius R of a monitoring station. For each polluting source si, a domain variable C_{si} has been created. This will contain the identification number of the source in correspondence of which the monitoring station controlling the source si is placed. The domain for C_{si} is therefore composed of the sources which are located within a range R centred on si

$$domC_{si} = \{j \mid S_j \; in \; a \; range \; R \; centred \; on \; S_i\}$$

The constraints imposing that each source must be controlled by a monitoring station are quite simple. For each pair of source si, sj they state that:

if two variables have the same element in their domains then they must have the same value

$$domC_{si} \cap domC_{sj} \Longrightarrow C_{si} =_C C_{sj}$$

The cost function is given by the number of different values given to the C_{si} variables. The predicate which builds the cost function constitutes a good example of the ECL^{i}PS^e expressive power; it is:

```
state_cost([Cs1, Cs2, ..., Csn], Cost]):-
   prune_instances([Cs1, Cs2, ..., Csn], Pruned),
   length(Pruned, Cost).
```

where `prune_instances/2` and `length/2` are both built-in predicates; the first one produces a list without any duplicated element, the second one calculates the length of a list.

The above described constraints well represent the problem, in fact we have obtained low execution times even with a high number of sources. Monitoring is an optimization problem with a small search space. Its CLP modelling has turned out to be efficient and intuitive.

The Detection problem

The Detection procedure compares the concentration values collected by a monitoring campaign with the ones predicted by the hydrodynamic model. Its aim is in detecting the polluting sources which are emitting more than they have declared. The principle which underlies such procedure is the following: assuming that the hydrodynamic model simulates correctly the hydrodynamic behaviour of the Lagoon, if the actual values differs from the values forecasted by the hydrodynamic model, then some sources are emitting more than they have declared.

The collected data are in the form (X, Y, Act_val) where Act_val is the concentration value (for a given substance) in the point (X, Y). For each of such points the constraint stated is:

$$Act\_val - G \leq f_{s1}(X,Y) * D_1 + \cdots + f_{sN}(X,Y) * D_N \leq Act\_val + G$$

where:

$f_{si}(X,Y)$ is the concentration factor (the one given by the hydrodynamic model) for the source si in the point (X,Y)

$s1, \ldots, SN$ are the active sources

D_{si} is a domain variable which represents the actual emission for the source si. Supposing that it is not likely that a source may emit less than it has been declared, we have stated the domain for the D_{si} variables to be between $e_{si,Sub}$ and $3 * e_{si,Sub}$, being $e_{si,Sub}$ the declared value for the source si.

G is a "tolerance value" which takes care of possible deviations for the data calculated by the hydrodynamic model.

Detection is a search problem. It looks for a set of emission values that may explain the actual concentration values. The role of the above described constraints is not in finding a precise single value for the D_{si} variables, but in reducing their domains to obtain the emission values intervals which may cause the measured concentration values.

Detection gives as results two lists of sources:

Lying_sources: contains the sources whose actual emissions are *surely* higher than the declared one. The minimum value in the domain for the D_{si} variables associate to such sources is higher than the declared value.

Maybe_lying_sources: contains some sources that are *likely* to be emitting more than they have declared. The minimum value in the domain for the D_{si} variables associate to such sources corresponds to the declared one, while the maximum value is higher.

Execution times have proved to be very low.

3.2 Some general remarks

Our search for an efficient representation of the problems composing the DSS has been driven by the need to obtain acceptable execution times. The indication regarding the limit under which an execution time can be considered "acceptable" has been indicated by the end users: the Water Magistracy staff. Their opinion is that computation times higher than a few hours (two or three at the most) should not be considered. When the problems are very complex and have a large search space, just the joint action of constraints and parallelism can obtain reasonable execution times.

One of the weakness of the CLP approach is, undoubtedly, the sensitivity to data and therefore the impossibility to predict execution times. We have often incurred into sets

of data which caused high execution times even with a problem formalization that, up to then, had been considered efficient enough. Anyway, an accurate analysis of the nature of such data has helped us to point out some principles which, translated into pre-computation rules or further constraints, have improved the problem modelling (the pre-computation rules for the Relocation problem have been generated in such a way).

The declarative nature of the ECL^{i}PS^e language has made the introduction of additional constraints in the programs an easy task. ECL^{i}PS^e makes the expression of constraints as simple as possible, and the wide range of powerful tools at the user's disposal can help to obtain remarkable results in terms of performance even with very complex problems.

The expressive power of ECL^{i}PS^e is undoubtedly valuable, but the best way of achieving an efficient computation for a problem consists in finding an appropriate formalization of it in terms of variables and constraints. The complexity of such a task is not ascribable to ECL^{i}PS^e or to any other programming language, but it arises from the more general and complex field of modelling. We have often been forced to restate our problems and the improvements obtained, when the new formalization has proved to be more suitable than the previous one, have sometimes turned out to be remarkable.

3.3 A foreseeable evolution for the DSS

The realization of the Venice Lagoon DSS within the APPLAUSE project has been a means of assessing the parallel programming system ECL^{i}PS^e. Nevertheless, our efforts have been aimed at building a prototype which reflects the topics of the operative environment to which it is dedicated. The prototype version of the DSS considers 12 different polluting substances and up to 80 polluting sources. Even though, for legal reasons, the Venice Water Magistracy has to trace each discharge in the Lagoon (at the moment something as 2500 sources are traced), just a few tens of these, the most important and dangerous ones, are taken into consideration by the Water Magistracy for the Lagoon safeguarding. The dimensioning of the prototype is, therefore, not so far from considering a realistic scenario.

Thinking of a possible evolution for the prototype, we have considered a two-level structure to raise the number of polluting sources considered by the DSS. This two-level approach consists in grouping a number (20 or 30) of real polluting sources (Low Level Sources) by locality, to constitute a "virtual" polluting source (High Level Source). The data relative to the High Level Sources are intended to summarize the polluting contribution of the composing Low Level Sources: the high level emissions are given by the sum of the low level emissions, while the high level costs are the average of the low level costs. The logic core of the DSS should remain essentially unchanged, but for the Reduction and Relocation problems (the most time-consuming ones) two levels of computations are allowed. A first level of computation considers just the High Level Sources and obtains the percentage of reduction or relocation for the emissions. The second level, if activated, redistributes the results obtained for a given High Level Source on the Low Level Sources which compose it. In our context, "to redistribute" a percentage of reduction (relocation), relative to a High Level Source, means to find a reduction (relocation) plan for the emissions of the composing Low Level Sources; the sum of the quantities reduced (relocated) on the Low Level Sources must be equal to the global reduction (relocation) indicated from

the first level of the computation. We have created a first version of the two procedures (*Low-Level-Relocation* and *Low-Level_Reduction*) which have to perform the redistribution. Both are optimization problems in that they minimize the cost arising from the reduction (relocation) plan. *Low-Level-Relocation* and *Low-Level_Reduction* are a simplified version of the correspondent high-level problems; they don't deal with concentration factors nor violation points and, therefore, a low number of constraints are involved. The simplifications introduced in *Low-Level-Relocation* have made it a linear problem and we have obtained an efficient solution by using the simplex method adopted by the ECL^{i}PS^e rational solver.

The implementation of the two-level structure has been just outlined during the Applause project, but it constitutes the guideline for a further evolution of the Venice Lagoon DSS.

4 Parallelization

Three distinct modes of parallelism are supported by the ECL^{i}PS^e environment: OR-parallelism, independent AND-parallelism and data parallelism. Our experience is mainly related to data parallelism which turned out to meet the requirements of our application. The DSS logic core description, which has been supplied in the previous section, makes it clear that not all the DSS modules needed a parallel approach. As for Detection, Monitoring and Analysis, the small search space (specially Monitoring), the nature of the problems and an efficient representation of these have allowed to obtain satisfactory performances even in the sequential version. So we have invested our efforts in exploiting parallelism in the Reduction and Relocation procedures, whose large search spaces required the joint action of constraints and parallel executions.

4.1 "Our parallelism"

Reduction and Relocation essentially consist of constrained search and optimization tasks; they compute the reduction/relocation plan for the source polluting emissions in order to bring the concentration values back to an allowed bound at the minimum cost. For both these procedures we have a two-level parallelism: a larger grain one, which originates a parallel computation for the substances considered by the DSS:

```
par_member(Substances, List)
```

and a finer one on the domain of domain variables representing the percentages of reduction/relocation:

```
par_indomain(Dom_variables).
```

The size of the search space for these problems is quite large, and the benefits coming from parallel annotations grow when the number of pollutant sources increases. The graph shown below refers to the Reduction problem; it displays good speed-ups (even a *super-linear speed-up* with 80 sources) arising from the independence of computations and a coarse-grained parallelism which outweigh the overhead of starting parallel processes. Although the *Relocation* problem is more complex than the *Reduction* one, its parallel

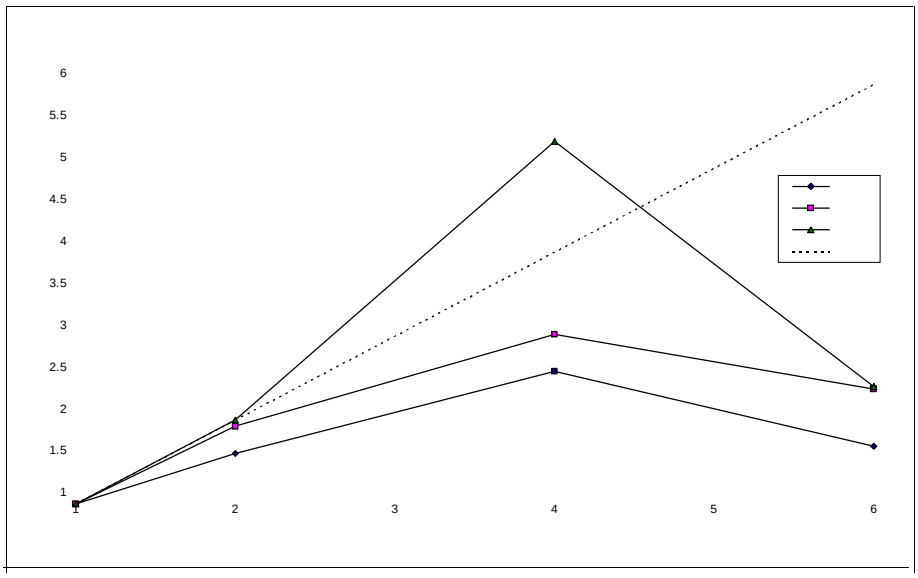


Figure 4.1: Speed-ups for the reduction problem

implementation has shown a speed-up which is closer to being linear. The speed-up figure for the reduction problem is given in figure 4.1.

4.2 Some general remarks about parallelism

When the APPLAUSE project started, we were perfectly aware of the importance of exploiting the benefits arising from parallelism, anyway the first most natural formulation of the modules composing the DSS was a sequential implementation. The following task consisted in introducing parallelism. The migration from the sequential version towards a parallel one required a minimal effort, in fact ECL^{i}PS^e enables the programmer to express parallelism very easily, simply using parallel predicates or annotations.

To introduce data-parallelism we didn't have to rewrite the code, it was sufficient to substitute some sequential predicates (`member/2`, `indomain/2`) with the parallel ones (`par_member/2`, `par_indomain/2`). In fact this data-parallelism is exploiting OR-parallelism of ECL^{i}PS^e.

In this way we obtained a parallel version for our application; a noteworthy effort has then been required to tune the application to obtain an efficient parallel version, in order to best exploit the benefits of parallelism.

ECL^{i}PS^e has been planned to support coarse-grained parallelism, so the most important problem was to estimate the ideal grain size for our modules. Our first approach to this problem was a "trial and test" approach, in fact we started introducing parallelism at the highest level (`substances`) and then we tried to add finer grained parallelism until the growing overhead caused the parallel execution to become inefficient. Enhancing our experience in parallelization issues, it has been more and more easy to identify the points where it was convenient to introduce parallelism, so some useless and time-consuming attempts have been avoided.

ECL^{i}PS^e allows the user to define the number of processes (also called workers) that jointly

execute the program in parallel; on this topic, our experience validates the suggestion given by the language developers: the number of workers should match the number of physical processors available on the machine.

5 Performance debugging and optimization

The whole DSS development has been characterized by a continuous search for performance improvements. To obtain such a result we have investigated about a number of factors which might affect the computation, ranging from considering pre-computation rules, to testing the effects of the different search algorithms provided by ECL^{i}PS^e. In the following we report about our experience in this optimization process.

5.1 About pre-computation

The pre-computation rules often have originated by the analysis of sets of data which caused particularly high computation times. Such data helped us in detecting some logical rules which, at first sight, had not been pointed out or considered important for the formalization of the problem.

There are two examples of pre-computation rules relative to the Reduction and Relocation problems:

The involved-sources rule

The principle which underlies this rule is valid for both the Reduction and the Relocation problems. It is very simple and sounds as follows: "If the emissions of a source S don't affect any violation point (X, Y) (this is true if the concentration factor for the source S is equal to zero in every point where the concentration value is higher than the allowed bound) then, reducing the emissions of the source S, no reduction of the concentration values in the violation points will follow".

In the first version of Reduction and Relocation procedures we created a domain variable for each source considered by the DSS, but we soon realized that, if the number of the violation points were low, the execution times were often high. Such a result is explainable considering that with a low number of violation points, we had a low number of constraints (as many as the number of violation points present) and a large search space (the number of sources is the exponent for the cardinality of the search space both for Reduction and Relocation). To efficiently deal with a low number of violation points, we have created a predicate `involved_source/3` which, before stating the constraints and entering the minimization phase, inserts in a list only the sources whose emissions affects at least one violation point. The computation time for this predicate is negligible, but the effects on the cardinality of the search space are remarkable.

As an example, using the involved-sources rule in the Reduction problem: with 40 polluting sources considered by the system and 4 violation points, the cardinality of the search

space has been reduced from 11^{40} to 11^8, where 8 is the length of the list returned by the `involved_source/3` predicate.

The non-solvable points rule

This second rule affects the Relocation problem. The principle on which the rule is based is: "A violation point can't be solved by relocating a quantity equal or greater than the bound (for a given substance) from an active source to an inactive one. In fact, in such a case, the inactive-source would, in turn, become a violation point". Such a consideration has allowed to reduce, a priori, the domain of the variables $Perc_{si}$, representing the percentage of emission that must be re-routed for the active source si. In a first version of the Relocation problem, each $Perc_{si}$ variable had the interval between 0 and 10 as domain; after inserting the non-solvable points rule, the maximum value for the domain ($Max\_dom\_Perc_{si}$) is calculated as follows:

$$Max\_dom\_Perc_{si} = Bound * 10/e_{si,Sub}$$

(where $e_{si,Sub}$ is the quantity of the substance Sub emitted by the source si)

in fact the disequation

$$e_{si,Sub} * Max\_dom\_Perc_{si} * 10/100 < Bound$$

must be verified. This, in a way, can be seen as pruning the domain "from the top".

An accurate analysis of the above mentioned principle has allowed us to prune, furthermore, the domain of the $Perc_{si}$ variables from the bottom. The reasoning is the following: the concentration value ($Conc\_val$) in a point (X, Y) is given by

$$Conc\_val(X, Y) = f_{s1}(X, Y) * e_{s1,Sub} + \cdots + f_{sN}(X, Y) * e_{sN,Sub}$$

that is the sum of the contributions arising from the emitting sources whose concentration factor in the point (X, Y) is not zero. If the contribution of a single source si in point (X, Y) ($Conc\_Si(X, Y)$) is greater than the allowed Bound, then the emissions of the source si have to be relocated by a percentage which is equal or greater than:

$$Min\_dom\_Perc_{si} = (Conc\_Si(X, Y) - Bound) * 100)/(Conc\_Si(X, Y) * 10)$$

$Min\_dom\_Perc_{si}$ is the minimum value for the domain of the $Perc_{si}$.

If for a domain variable $Perc_{si}$, it results that

$$Min\_dom\_Perc_{si} > Max\_dom\_Perc_{si}$$

then the point (X, Y) cannot be solved by the Relocation strategy, but the only way to reduce its concentration value is by the reduction of the emissions.

$Min\_dom\_Perc_{si}$ and $Max\_dom\_Perc_{Si}$ are worked out before stating the constraint and entering the minimization phase, so the *non-solvable points* rule immediately detects sets of violation points for which there is not a solution by Relocation.

5.2 User-defined constraints

The non-linear nature of the constraints defining the Relocation problem has made necessary to create a predicate (`qeq/3`), in order to force the propagation mechanism in the non-linear terms.

As stated in 3.1, the Relocation problem involves two different kinds of domain variables:

- the $Perc_{si}$ variables which represent the percentage of emission relocation for the source si.
- the I_{si} variables that indicate which is the inactive source to which part of the emissions of the active source si must be relocated.

Such variables are used in the definition of both the constraints and the cost function. Let's consider the cost function. This is defined as:

$$Perc_{s1} * C_{s1,Is1} + \cdots + Perc_{sN} * C_{sN,IsN}$$

where $C_{si,Isi}$ is a domain variable representing the linear cost to relocate the 10% of the emission of the source si to the inactive source indicated by the variable I_{si}. The domain for $C_{si,Isi}$ is composed by all the relocation costs associated to the active source si (there exists a relocation cost for each inactive source). Each term of the cost function is therefore given by the product between two domain variables. In such a case (non-linear terms) the ECL^{i}PS^e automatic propagation mechanism doesn't work, that is the updates on the domain of a variable don't automatically cause the updates on the domains of the other variables linked to the modified one by the constraints. `qeq/3` aims at forcing such a propagation. The principles on which `qeq/3` is based are the following:

if A and B are two domain variables and $C = A * B$, then C is, in turn, a domain variable $A, B, C \geq 0$ and

1. $minC \geq minA * minB$ ($minA$, $minB$, $minC$ are the minimum values for the variables A, B, and C respectively)
2. $maxC \leq maxA * maxB$ ($maxA$, $maxB$, $maxC$ are the maximum values for the variables A, B and C respectively)

so the values higher than $maxC$ and lower than $minC$ must be eliminated from the domain of C.

Considering that $C = A * B$ can be expressed also as $A = C/B$ and $B = C/A$, the following relations can be stated:

3. $maxA \leq maxC/minB$ (and $maxB \leq maxC/minA$)

4. $minA \geq minC/maxB$ (and $minB \geq minC/maxA$)

From relations 1. 2. 3. and 4., it is manifest that each change involving the domain limits of a variable affects the domains of the other variables, so the `qeq/3` predicate must be woken as soon as a domain variable limit is updated.

In the following the definition of the predicate `qeq/3` is given. The predicates `propmax/5`, `propmin/5` and `ntimes/3` are user-defined predicates, the remaining ones are ECL^{i}PS^e built-in predicates. `propmax/5` and `propmin/5` refers to the relations 3. and 4. respectively, while `ntimes/3` refers to the case in which the variables are equal to zero.

```
qeq(Perc,CP,C) :-
   dvar_domain(Perc,DomPerc),
   dvar_domain(CP,DomCP),
   dvar_domain(C,DomC),
   dom_range(DomPerc,MinPerc,MaxPerc),
   dom_range(DomCP,MinCP,MaxCP),
   dom_range(DomC,MinC,MaxC),
   Min is MinPerc*MinCP,
   Max is MaxPerc*MaxCP,
   (Min > MinC ->
      dvar_remove_smaller(C,Min),
      Upd = 1
   ;
      true
   ),
   (Max < MaxC ->
      dvar_remove_greater(C,Max),
      Upd = 1
   ;
      true
   ),
   propmax(MaxC,MinPerc,MaxCP,CP,Upd),
   propmax(MaxC,MinCP,MaxPerc,Perc,Upd),
   propmin(MinC,MinPerc,MaxCP,Perc,Upd),
   propmin(MinC,MinCP,MaxPerc,CP,Upd),
   (nonvar(Upd) ->
      qeq(Perc,CP,C)
   ;
      Vars = p(C,Perc,CP),
      term_variables(Vars,VL),
      length(VL,N),
      (N = 3 ->
         make_suspension(qeq(Perc,CP,C),4,Susp),
         insert_suspension(Vars,Susp,min of fd,fd),
         insert_suspension(Vars,Susp,max of fd,fd)
```

```
    ;
    N = 2 ->
       ntimes(Perc,CP,C)
    ;
       times(Perc,CP,C)
    ),
    wake
).
```

The introduction of `qeq/3` has proved to be determinant in obtaining acceptable execution times. In the previous version, when the propagation for the non-linear terms was not active, the constraints didn't succeed in reducing dynamically the domain variables and this resulted in ineffective performances.

5.3 min_max and minimize

The logic core composing the Venice Lagoon DSS deals mainly with optimization problems, so we have made an intensive use of the built-in predicates `minimize/2` and `min_max/2`, which ECL^{i}PS^e provides for minimization purposes. `min_max` and `minimize` have different search strategies (local backtracking for `minimize` vs. recomputation for `min_max`), and we have tested both of them in each of our optimization problems to choose the most effective one. As for our experience, `min_max` has always proved to provide the best performances.

We have found the `min_max/5` and `minimize/5` predicates very useful. These predicates allow to define a minimum and a maximum value for the cost function and a percentage Perc of tolerance. The predicates with arity 5 consider equivalent the solutions within the range of Perc% and, therefore, start the search for the next better solution with a minimized value Perc% less than the previously found one. Even by indicating low percentage of tolerance (5%) we have obtained remarkable improvements for the execution times. A 5% approximation, on the other hand, doesn't affect the relevance of the optimization process, in fact, in the context of such complex modelling problems, it is often sufficient to find a solution which is close enough to the best one instead of searching for the optimum.

6 Conclusions

The development of the Venice Lagoon DSS has constituted one of the test benches on which the APPLAUSE project aimed at assessing the suitability of ECL^{i}PS^e as a valid programming environment. ECL^{i}PS^e has been designed to exploit the combined potential of parallelism and Constraint Logic Programming. The joint action of both such approaches was promising in efficiently tackling the complex field of the environmental DSS. The decision-making activity the DSS intends to support is affected by a relevant number of different factors: regulations, laws, hydrodynamic principles, social and economical interests. As a consequence of such a complexity, the problems composing the DSS logic core have turned out to be mainly characterized by a non-linear nature which makes it necessary to consider a large amount of possible solutions. The constraint logic program-

ming paradigm and its declarative nature have eased the knowledge representation, while parallelism has allowed to face the involved large search spaces.

The DSS development has required a noteworthy effort, but it has led us to gather experience both in knowledge based systems and constraints logic programming.

The search for an efficient modelling of the DSS problems in terms of variables and constraints has proved to be the most challenging activity during the development process. We have found that the constraint handling of ECL^{i}PS^e and its expressive power well support the formalization of even the most complex relations and knowledge. The use of parallelism has been fundamental to deal with search spaces which were not pruned heavily enough by the constraints. ECL^{i}PS^e enables the programmers to express parallelism very easily, but a tuning activity has been necessary to exploit it at its best.

As for the user-interface development, ECL^{i}PS^e provides the access to an external public domain graphic tool: Tcl/Tk. Tcl/Tk has been especially planned to support the creation of graphic interfaces, so it has been quite easy to supply the DSS with an efficient and user-friendly interface.

Concluding, our contribution for the assessment of ECL^{i}PS^e as a programming environment results in a positive judgement. Of course, parallel CLP may not be adequate for some kinds of problems, but the winning strategy of ECL^{i}PS^e consists in allowing the use of the most adequate approach for the problems by integrating or interfacing existing systems.

Chapter 6.

Decision Support in Molecular Biology

Chris Rawlings and Dominic Clark

1 Problem Description - Predicting Protein Structure

Proteins mediate most biological activities in the body including respiration, cell growth and cell differentiation. The success of many aspects of medical science and the biotechnology industry is dependent on a detailed understanding of the structure of proteins and in particular how changes in structure influence the function and biological role of the protein in the cell. Many aspects of cancer research also rely on knowledge about protein structure and function at the molecular level.

In general, a complete characterization of the function of a protein depends upon knowing its precise three dimensional atomic structure. Solving the 3D structure by x-ray crystallography or by nuclear magnetic resonance is, however, time consuming and often technically very difficult. Methods for predicting the structure of a protein from the sequence of amino acids therefore has a significant potential for resolving the cost and complexity of protein crystallography.

The most successful developments in protein structure prediction have used model building [BSST87]. These methods rely on aligning the amino acid sequence of the protein of unknown structure with another from a protein with known 3D structure and using this as the structural model. However, although prediction based on model building from an analogous 3D-structure is likely to yield the most reliable structures there is a requirement that the amino sequences be very similar (25% or more). Unfortunately, for many of the proteins in the protein sequence databases, there is no analogous sequence present in the protein structure database. Alternative methods for predicting protein structure from sequence data that complement model-building are therefore required.

1.1 Protein Topology Prediction

Protein topology denotes a level of protein structural organization that is intermediate between the secondary and tertiary levels. A topological description uses spatial and order relationships among protein secondary structures (α-helices and β-strands in β-sheets) (Figure 1.1).

A prediction of the topological structure of a protein can be used to identify proteins with a similar topological structure where the similarity is not detectable at the sequence level and in the more general case, should provide sufficient structural information to guide the selection of experimental investigations to verify the prediction or to help elucidate its biological function. Figure 1.2 illustrates how we represent the structure of all β and α/β proteins at the topological level. The essentially planar structure of β-sheets enables a simple list data structure to be used.

Unlike β proteins whose general folding architecture has been accepted since the first public databank of protein structure [BKW$^+$77] and has come to be a fundamental part of the way in which information about β-structures is stored in public domain databases, a general architecture (or parameterization) of the α-helical globule has only recently been proposed[MF88]. In this description the α-helical globule can be modelled by fitting the core regions of helices on the ribs of quasi-spherical polyhedra (henceforth deltahedra

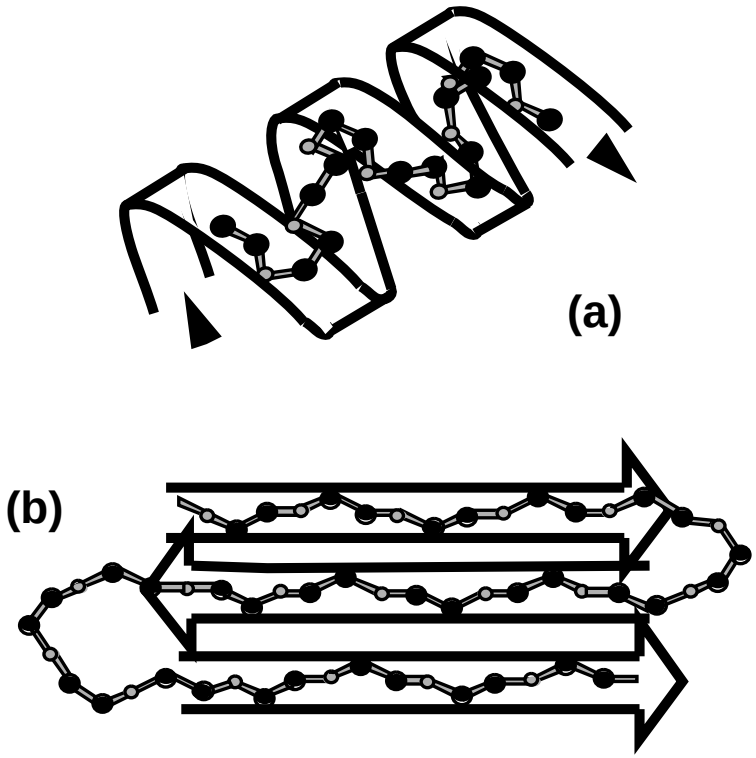
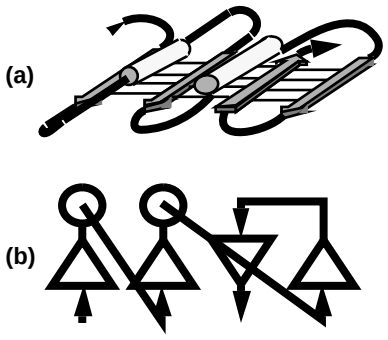


Figure 1.1: **Regular local protein structures.**
A schematic representation of the two most important classes of secondary structure observed in proteins: (a) α-helix and (b) a β-pleated sheet composed of antiparallel β-strands. Parallel β-sheets are also found, but are not shown here.

(Figure 1.3) such that only one helix occupies any one vertex. The cores of individual helices can then be though of as packing along the ribs of these polyhedra.

The prediction of protein topology relies on the availability of an assignment of secondary structure that must be predicted from the protein amino acid sequence. Despite well known problems with the accuracy of secondary structure prediction methods, it is nevertheless possible to predict topology from secondary structure using rules of protein folding. So long as the secondary structure of the protein is accurate at a segmental level (i.e. the predicted secondary structural regions roughly overlap the true ones allowing some error in the specification of termini) many topological folding rules (relating to handedness of connections, orientation, strand positions etc.) can still be applied to the predicted secondary structure. Furthermore, uncertainty in secondary structural assignments can be accommodated to some extent by relaxing the applicability conditions of some folding rules.

In this chapter we describe three different programs that we have developed using parallel constraint logic programming tools to experiment with new methods for predicting the topology of proteins. The first, CBS1e addresses the problem of predicting the β-sheet topology of α/β proteins, the second CBS2e extends CBS1e to deal explicitly with the problem of representing soft (non-categorical) constraints and the third, HFE, explores the problem of representing the fold topology of all-α proteins.




```
(c) [position(1,1,u,u)
    ,position(2,2,u,r)
    ,position(4,3,u,r)
    ,position(3,4,d,r)]
```

Figure 1.2: **Representation of the topological structure of an α/β sheet:** Panels (a) and (b) show alternative diagrammatic representations of protein topology. In panel (a) a simple β-sheet with α-helices packed against the upper surface is shown with the α-helices drawn as cylinders and the β-strands as solid arrows which indicate their relative orientations in the sheet. Such sheets are often referred to as α/β-sheets. Panel (b) shows the same structure re-drawn in a planar topology diagram with β-strands drawn as triangles and α-helices as circles. In (b) the sheet has been drawn looking from above and then subjected to a single (topologically neutral) rotation so that the positions of the strands align. In panel (c), the list of position/4 terms for the sheet show how this topology is represented in CBS1e and CBS2e.

2 Qualification

2.1 α/β Sheets

The principal difficulty when predicting protein topology from secondary structure is that, in the absence of other constraining information, a vast number of topological conformations can potentially result from a single set of secondary structure assignments. Specifically, we showed in [DSR91] that, after making simplifying assumptions and considering only:

- neighbourhood relations between strands
- left or right handed parallel (crossover) connections
- one type of antiparallel (hairpin) connection between strands

For an α/β-sheet of n strands ($n > 1$) the number of possible strand topologies, p, is given by:

$$p = n!(3^{n-1})/2 \tag{0.1}$$

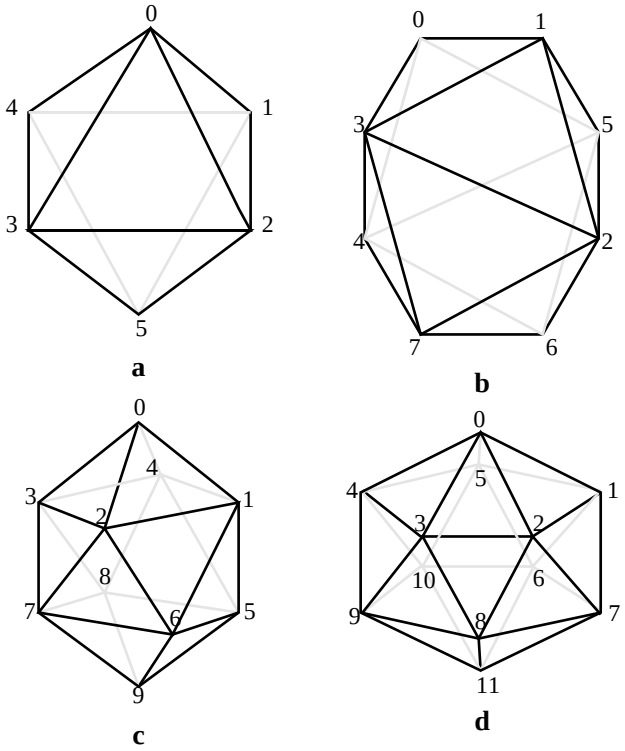


Figure 1.3: **Representation of the topological structure an α-helical globule:** A deltahedral representation of α-helical proteins redrawn from [MF88] for packing of globular helical motifs with 3 (a), 4 (b), 5 (c) or 6 (d) helices. All deltahedra are composed of equilateral triangles.

No. of strands	2	3	4	5	6	...	10
No of topologies	4	48	768	15,360	368,640	...	$4.75x10^{11}$

Table 2.1: Values for p and n derived from equation 2

If left and right handed hairpin (antiparallel) connections are also distinguished, adding a fourth type of connection, then the number of topologies, p, for a given sheet of n strands is given by:

$$p = n!(4^{n-1})/2 \tag{0.2}$$

Typical values for equation 0.2 are given in table 2.1.

The first program to explicitly view protein topology prediction as constraint satisfaction and to use techniques from Logic Programming was CBS1 (Constraint Based Search, version 1, [DSR91]).

CBS1 demonstrated that using the LP language Prolog it was possible to succinctly implement both protein topological folding rules and a protein topology prediction algorithm based on constraint satisfaction. Two key factors suggested that CBS1 should be extended and that it would be important for the basic approach be made more computationally effi-

No. helices	No. of windings	Elapsed time/s
3	16	0.136
4	816	3.200
5	19200	89.600
6	96000	410.000

Table 2.2: Number of windings as a function of number of helices versus elapsed times on 1 SUN Sparc2 processor

cient. Firstly, in a practical protein topology prediction system it is necessary to consider either multiple secondary structure predictions or to use more detailed representations of the protein structure. Either of these developments would greatly increase the combinatorics of the search space. Secondly, because exceptions can be found to many of the proposed general protein folding rules the approach to constraint satisfaction in CBS1 needed to be extended from one which views all constraints as categorical to one which manages both categorical and partial (uncertain) constraints. Again it was anticipated that there would be performance penalties incurred when extending CBS1e to accommodate uncertain constraints.

It was decided that parallel CLP would be an ideal environment to develop the new versions of the protein topology prediction programs. The main reason for thinking that parallelism would be useful was the belief that the known protein folding rules would not fully constrain the search space and thus there would always be a considerable amount of search to find topologies that were consistent with the data and constraints.

2.2 all-α proteins

Quantifying the problem space for the folding of α-helical bundles from first principles is not so straightforward. We therefore chose to determine the size of the problem space empirically from an implementation of the deltahedral framework as a CLP program. The problem space is characterised as the number of ways in which the protein chain can wind around the deltahedral framework.

Table 2.2 shows the number of windings as a function of the number of helices along with elapsed times on a Sun MP/630 with one processor (Sparc2) and in Figure 2.1 it is clear that the search space grows less than exponentially with respect to the number of helices in the globule. Although the execution times for this problem were not inordinately long, it was considered important that any developments that should stem from this work should be founded on an efficient and scalable implementation.

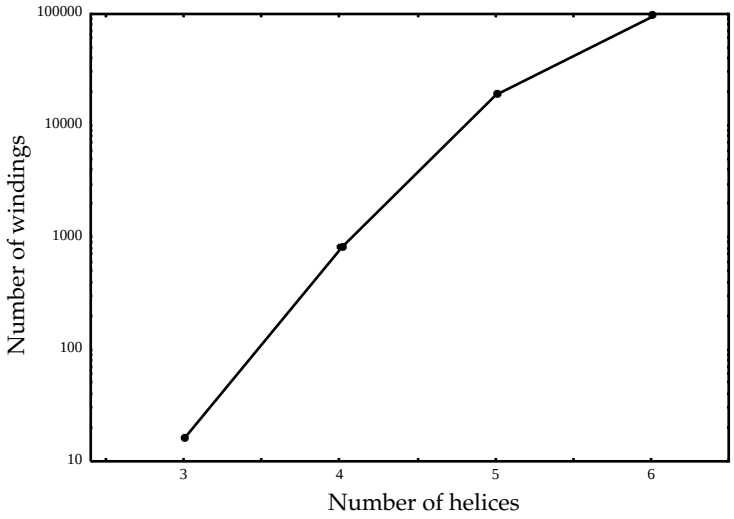


Figure 2.1: **Quantifying the problems space for α-helix folding**
The number of possible windings of the protein backbone as determined by the HFE program. The increase in the problem space is less than exponential with respect to the number of helices.

3 Constraint Modelling and Prototyping

3.1 α/β sheets topology - CBS1e/2e

The topology of a β sheet of N strands is represented in CBS1e as a list of N terms each of the form `position(F,Y,Z,C)`, where F is the position of that strand in the sheet template in the range [1,N], Y refers to the sequential position of the strand in the sequence in the range [1,N], Z is the orientation of the strand (up or down) and C is the chirality of the connection to the preceding strand (left, right or undefined). See Figure 1.2c.

Search strategy and use of CLP features

In general the most natural was to express a program in a CLP language is to (a) define the hypothesis space, (b) specify the constraints on that hypothesis space and (c) initiate a search (or labelling) procedure. In the following, we distinguish between logical constraints which are part of the problem specification and empirical constraints which are independent of the problem specification per se, but which are selected by the user or program to constrain the hypothesis space. The empirical protein folding constraints used in CBS1e come from a paper in which Taylor and Green predicted the topology of a cation transporting ATPase (an enzyme involved in the transport of magnesium or calcium ions) [TG89]. They are listed in Table 3.1. The top level goals of CBS1e are shown in Figure 3.1.

The CBS1e algorithm operates as follows

1. Creation of a sheet template with the appropriate number of strand slots. The call `create_strands_template_with_N_slots(4,Solution)` causes ECL^{i}PS^e to unify the logical variable `Solution` with a list of finite domain terms which represents

Name	Description
c1	For parallel pairs of β-strands, β-α-β and β-coil-β connections are right handed.
c2	The initial β-strand in the amino acid sequence is not an edge strand in the sheet.
c3	Only one change in winding direction occurs.
c4	The β-strands associated with the conserved patterns lie adjacent in the sheet.
c5	All strands lie parallel in the β-sheet.
c6	Unconserved strands are at the edge of the sheet.
f2	Parallel β-coil-β connections should contain at least 10 residues in the coil.

Table 3.1: Protein Folding Constraints used by CBS1e

the possible assignment of (in this case) a β-sheet with 4 strands. This is thus the situation before the finite domains (variable lists of possible values of a variable as a list in curly braces "{}") have been reduced by the application of any constraints:

```
Solution=[
position(_{1,2,3,4},_{1,2,3,4},_{u,d}, _{l,r,a,u}),
position(_{1,2,3,4},_{1,2,3,4},_{u,d}, _{l,r,a,u}),
position(_{1,2,3,4},_{1,2,3,4},_{u,d}, _{l,r,a,u}),
position(_{1,2,3,4},_{1,2,3,4},_{u,d}, _{l,r,a,u})
].
```

2. Application of logical (structural) constraints. In this formulation of β-sheet topology the first strand is arbitrarily assigned to the left hand side the sheet and oriented up (with undefined handedness). This constrains two independent degrees of rotational invariance. The other strands have either left or right handed connections. A further logical constraint ensures that all strands are assigned different template positions and that the solution gives the strand numbers in sequence order. The hypothesis space is therefore reduced through propagation to:

```
Solution=[
position(_{1,2},     1,   u,      u),
position(_{1,2,3,4}, 2, _{u,d}, _{l,r,a}),
position(_{1,2,3,4}, 3, _{u,d}, _{l,r,a}),
position(_{1,2,3,4}, 4, _{u,d}, _{l,r,a})]
```

   Note that this provides the most general representation of the hypothesis/solution space.

3. Constraint Ordering.
   In general the order in which constraints are applied in a search algorithm can affect the performance of that algorithm. In CBS1e, the empirical constraints are ordered such that those which are non-deterministic (contain choice points) are evaluated last, in this case c3. The rationale for this is to maximize the amount of resolution (propagation) that the system performs to minimize the search space. To take a trivial example, if A and B are finite domain variables {1,2,3} and `2 * A #= B`, then ECL^{i}PS^e immediately infers A=1, B=2. There is no search.

```
generate_topologies(N,Constraint_list,Topologies_List):-
        create_strands_template_with_N_slots(N,Topologies_List),
        apply_unary_logical_constraints(Topologies_List),
        extract_lists(Topologies_List,List_of_lists),
        apply_set_theoretic_logical_constraints(N,List_of_lists),
        order_empirical_constraints(Constraint_List,OC_List),
        apply_empirical_constraints(OC_list,N,Topologies_List,List_of_lists),
        instantiate(Topologies_List).

apply_unary_logical_constraints([]).

apply_unary_logical_constraints([position(F,Y,Z,C)|T]):-
        first_strand_is_orientation_up(Y,Z),
        first_strand_is_of_undefined_chirality(Y,C),
        apply_unary_logical_constraints(T).

apply_set_theoretic_logical_constraints(N,Lists):-
        all_different_template_positions(Lists),
        strictly_orderedYs(Lists),
        first_strand_is_on_the_left(N,Lists).
```

Figure 3.1: **The top level goal of CBS1e:**

The basic strategy exploits the constrain and generate paradigm of CLP i.e. defining the hypothesis space by producing a solution template, applying logical constraints to this template which are part of the problem structure, then ordering applying empirical constraints selected by the user or program which are not part of the problem structure then finally instantiating any solution templates with domain variables.

4. Application of Empirical Constraints
   The empirical constraints are then applied and their effects propagated through the search tree, with the following partial solution generated as constraints are added:

   With constraints c2 only (the first strand is non edge) we have:

```
Solution=[
position(  2,      1,   u,      u),
position(_{1,3,4}, 2, _{u,d}, _{l,r,a}),
position(_{1,3,4}, 3, _{u,d}, _{l,r,a}),
position(_{1,3,4}, 4, _{u,d}, _{l,r,a})]
```

   With constraints c5 (all strands are parallel), c1 (all parallel connections are right handed), and c2 we have:

```
Solution=[
position(2,        1, u, u),
position(_{1,3,4}, 2, u, r),
position(_{1,3,4}, 3, u, r),
position(_{1,3,4}, 4, u, r)]
```

   With constraints c1,c2, c5 and c3 (at most 1 change in winding direction) and only now introducing labelling/search, we have the following set of solutions:

```
Solution1=[
position(2, 1, u, u),
position(1, 2, u, r),
position(3, 3, u, r),
position(4, 4, u, r)];

Solution2=[
position(2, 1, u, u),
position(3, 2, u, r),
position(4, 3, u, r),
position(1, 4, u, r)];

Solution3=[
position(2, 1, u, u),
position(4, 2, u, r),
position(3, 3, u, r),
position(1, 4, u, r)]
```

There are four essential differences between the ECL^{i}PS^e specification of constraints and the Prolog specification in the prototype Prolog version (CBS1). The ECL^{i}PS^e representation of rule c3 can be found in Figure 3.2. This representation differs from the Prolog specification since (a) all predicates must be defined (e.g. `nonmonotonic/3`) rather than relying on negation as failure (`not(monotonic/3)`), (b) predicates with choice points can be defined as parallel (e.g. `changes_in_winding/2, monotonic/3, nonmonotonic/ 3`), and (c) using arithmetic constraint operators certain predicates can be made to 'delay'. In this case the predicates `monotonic/3` and `nonmonotonic/ 3` can be made to delay until all their arguments are ground either by explicit finite domain relational constraints, or through mode declarations (e.g. here commented out for `monotonic/3` and `nonmonotonic/3` where all three arguments are specified as ground).

In ECL^{i}PS^e, all equalities and disequalities are specified as arithmetic constraints which are not simply instantiated and then checked, but which are implemented using a look ahead inference rule. The predicate `nonmonotonic/3` is explicitly defined rather than relying on negation as failure on the predicate `monotonic/3`.

In ECL^{i}PS^e, the predicates with choice points (`changes_in_winding/2, monotonic/3` and `nonmontonic/3`) can be executed in parallel. Finally, the goal suspension mechanism specified either by the mode declaration for `monotonic/3` and `nonmonotonic/3` or by using arithmetic constraints on finite domains has the effect of causing the constraint to delay if called with the inappropriate instantiation. It will, however, subsequently be "woken up" and evaluated if the right instantiation of variables later occurs.

The standard execution strategy of Prolog like systems is the so called "fixed left to right depth-first" strategy in which predicates described in the body of a clause are executed in the order specified. This is fine for some aspects of programming however for constraint checking it imposes the additional burden upon the programmer of ensuring that all variables are of the right level of instantiation when checked (i.e. $A > 1$ will fail if A is variable). This often has the result of reducing the declarativeness of programs.

In ECL^{i}PS^e, the delay feature is a way of changing this execution model so that it depends on the state of instantiation of the arguments rather than their textual position in the source code.There are many potential advantages provided by this feature including

```
constraint(c3,PositionList):-
        max_changes_in_winding(Atmost),
        changes_in_winding(PositionList,Current),
        Current #<= Atmost.

max_changes_in_winding(1).

?- parallel changes_in_winding/2.
changes_in_winding([P1,P2,P3|T], 0 + Current):-
        monotonic(P1,P2,P3),
        changes_in_winding([P2,P3|T],Current).

changes_in_winding([P1,P2,P3|T], 1 + Current):-
        nonmonotonic(P1,P2,P3),
        changes_in_winding([P2,P3|T],Current).

changes_in_winding([],0).
changes_in_winding([_],0).

changes_in_winding([_,_],0).

?- parallel monotonic/3.
monotonic(P1,P2,P3):-
        P1 #> P2, P2 #> P3.

monotonic(P1,P2,P3):-
        P1 #< P2, P2 #< P3.

?- parallel nonmonotonic/3.
nonmonotonic(P1,P2,P3):-
        P1 #> P2, P2 #< P3.

nonmonotonic(P1,P2,P3):-
        P1 #< P2, P2 #> P3.
```

Figure 3.2: **ECL^{i}PS^e representation of rule c3:**

greater declarativeness and more efficient programs through simulated co- routining.

CBS2e

CBS2e is an extension of CBS1e that deals with weighted constraints. Its operation is similar to CBS1e except that:

The failure conditions of each constraint must be each specified in addition to the success conditions. Each constraint is also assigned a pair of values (weights or penalties) corresponding to the truth functional states of the constraint (i.e. a value for the constraint being true and a value for the constraint being false) which are each ≥ 0.

These weights come from our empirical analysis of 8 nucleotide binding proteins [DSR91].

Constraint	Penalty if false [0,8]	Penalty if true [0,8]
c1	8	0
c2	7	1
c3	6	2
c5	5	3
f2	6	1

Table 3.2: Weights for Topological Folding Constraints

Cost function definition

The cost assigned to any set of topological hypotheses is defined as the sum of the weights (penalties) corresponding to the truth conditional states of the constraints to which it corresponds. In Table 3.2, for example, the set of topologies for which all constraints are true would have a total cost of 9 assuming a simple cost function.It should be emphasized that although adoption of a linear cost function makes the assumptions that the constraint weights can be combined independently and additively, many more complex weighting procedures can be reduced, via the appropriate transformation, to a linear cost function (e.g. probabilities though the use of a function such as mod(-Klog(odds))).

Cost minimization

A minimum value for this penalty sum is then determined using a branch and bound search using the in built search predicate `minimize/3`. This algorithm operates by pruning all branches of the search tree for which the linear cost function cannot be less than the current minimum. Here it may be the case either that the value already exceeds the minimum or that the remaining choices are such that under no combination can a value less than the minimum be found.

Finally all topologies with this minimum value are generated using the same search procedure called without the minimization function but with the additional explicit constraint that the precise value of the cost function be equal to the minimum already determined.

More details of CBS2e and it's implementation can be found in references [CRS+93a, CRS+93b].

3.2 all-helix bundle topology

The representation of α-helical bundles as deltahedra uses connectivity graphs where the qualitative spatial relationships in each deltahedra are represented as a set of connectivity relations (Figure 3.3). Here each deltahedra is represented as a set of clauses `conn(N,V,ListV)/3` where `N` is the number of helices in the globule, `V` is a vertex number and `ListV` are the vertices connected to that vertex.

The precise implementation of HFE uses both the finite domains mechanism of ECL^{i}PS^e, goal suspension, and a number of the built-in constraints. Figure 3.4 and Figure 3.5 show

```
conn(3,0,[1,2,3,4]).
conn(3,1,[0,2,4,5]).
conn(3,2,[0,1,3,5]).
conn(3,3,[0,2,4,5]).
conn(3,4,[0,1,3,5]).
conn(3,5,[1,2,3,4]).

conn(6,0,[1,2,3,4,5]).
conn(6,1,[0,2,5,6,7]).
conn(6,2,[0,1,3,7,8]).
conn(6,3,[0,2,4,8,9]).
conn(6,4,[0,3,5,9,10]).
conn(6,5,[0,1,4,6,10]).
conn(6,6,[1,5,7,10,11]).
conn(6,7,[1,2,6,8,11]).
conn(6,8,[2,3,7,9,11]).
conn(6,9,[3,4,8,10,11]).
conn(6,10,[4,5,6,9,11]).
conn(6,11,[6,7,8,9,10]).
```

```
conn(4,0,[1,3,4,5]).
conn(4,1,[0,2,5,3]).
conn(4,2,[1,3,5,6,7]).
conn(4,3,[0,1,2,4,7]).
conn(4,4,[0,3,5,6,7]).
conn(4,5,[0,1,2,4,6]).
conn(4,6,[2,4,5,7]).
conn(4,7,[2,3,4,6]).

conn(5,0,[1,2,3,4]).
conn(5,1,[0,2,4,5,6]).
conn(5,2,[0,1,3,6,7]).
conn(5,3,[0,2,4,7,8]).
conn(5,4,[0,1,3,5,8]).
conn(5,5,[1,4,6,8,9]).
conn(5,6,[1,2,5,7,9]).
conn(5,7,[2,3,6,8,9]).
conn(5,8,[3,4,5,7,9]).
conn(5,9,[5,6,7,8]).
```

Figure 3.3: **Connectivity matrices for deltahedra:**
Numbers refer to vertices as labelled in Figure 1.3. In `conn(+N,+V,+ListV)/3`, `N` is the number of helices in the globule, `V` is a vertex number and `ListV` are the vertices connected to that vertex. Thus, for example, in the six helical case, vertex 0 is connected to vertices 1, 2, 3, 4 and 5.

the top level goals which generate either all windings with short connections (a and b) or with any connections (c and d) either in a failure driven (a and c) or list collection style (b and d).

Intermediate models are also possible (e.g. at most one long loop extension), though these are arguably best modelled as constraints on the more general models in Figure 3.5d. As with other ECL^{i}PS^e programs produced at the ICRF under APPLAUSE the basic strategy is one of constrain and generate as embodied in the strategy of:

- defining the hypothesis space
- defining the structural constraints, and
- labelling/search

Windings and Motifs: defining the hypothesis space.

The test query is based on finding all motifs and windings for a particular helical bundle. The definition of motif taken is that implicit in [MF88]. Namely, a motif is a rotationally distinct assignment of helices to ribs in a deltahedra which (a) ignores connections between helices; (b) ignores the orientation of helices and (c) prohibits rotational, though not mirror, symmetries. The number of motifs associated with each deltahehra by Murzin and Finkelstein[MF88] is shown in Table 3.3.

```
(a)
all_windings_with_short_connections(N):-% N exists in {3,4,5,6}
        set_up_FDs(N,ListFD),
        apply_structural_constraints(N,ListFD),
        find_windings_with_short_connections(N,ListFD),
        write_out(ListFD),
        fail.
all_windings_with_short_connections(_).

(b)
all_windings_with_short_connections(N,ListWindings):-
        setof(Winding,one_winding_with_short_connections(N,Winding),ListWindings).

one_winding_with_short_connections(N,Winding):-
        set_up_FDs(N,ListFD),
        apply_structural_constraints(N,ListFD),
        find_windings_with_short_connections(N,ListFD),
        Winding = ListFD.
```

Figure 3.4: **Top Level goals (a) and (b) of HFE:**

No. Helices	No. Vertices	No. Motifs
3	6	2
4	8	10
5	10	10
6	12	8

Table 3.3: Number of Motifs as a function of the Number of helices and vertices (Murzin and Finkelstein, 1988)

As the number of motifs is quite small, categorizing helix folding in terms of motifs provides a useful basis for grouping the folded structures. Several definitions of the concepts of winding are possible. In this document a winding is viewed as an ordered set (or list) of distinct vertex numbers v1, v2 ...v2n for n helices, where v1 is the number of the vertex occupied by the n-terminus of the first helix, v2 is the vertex number of the vertex occupied by the c-terminus of the first helix, v3 is the vertex number of the vertex occupied by the n-terminus of the second helix and so on, the cardinality (or length) of the set being 2n for $n \in \{3, 4, 5, 6\}$ helices.

The ECL^{i}PS^e definition of the hypothesis space is based on the use of finite domain variables representing vertex numbers. Each winding is represented as an ordered list of 2n elements each element being a finite domain variable in the range `[0..m]` where `m` is 2n-1, following the vertex naming convention in Figure 1.3.

```
(c)
all_windings_with_any_connections(N):-
        set_up_FDs(N,ListFD),
        apply_structural_constraints(N,ListFD),
        find_windings_with_any_connections(N,ListFD),
        write_out(ListFD),
        fail.
all_windings_with_any_connections(_).

(d)
all_windings_with_any_connections(N,ListWindings):-
        setof(Winding,one_winding_with_any_connections(N,Winding),ListWindings).

one_winding_with_any_connections(N,Winding):-
        set_up_FDs(N,ListFD),
        apply_structural_constraints(N,ListFD),
        find_windings_with_any_connections(N,ListFD),
        Winding = ListFD.

apply_structural_constraints(N,ListFDVertices):-
        alldistinct(ListFDVertices),
        fix_first_helix(N,ListFDVertices).
```

Figure 3.5: **Top Level goals (c) and (d) of HFE:**

Thus for example, prior to any structural constraints being applied, the most general description of the hypothesis space of windings for 3 packed helices is given as:

```
Winding3 = [_fdV1{0,1,2,3,4,5}, _fdV2{0,1,2,3,4,5},
_fdV3{0,1,2,3,4,5}, _fdV4{0,1,2,3,4,5}, _fdV5{0,1,2,3,4,5},
_fdV6{0,1,2,3,4,5}].
```

Structural Constraints on Windings

Three types of structural constraints are applied to windings (Figure 3.6). Firstly, all vertices in a winding must be distinct. This is a very powerful constraint for pruning the hypothesis space using the consistency methods built into the ECL^{i}PS^e inference engine. Secondly, the position of the first helix is fixed along some rib to prohibit rotationally symmetric windings from co-occurring. The third type of structural constraint concerns adjacency. Given the definition of a winding above, it follows that there is a requirement for adjacency between pairs of vertices in the vertex list when the pair of vertices correspond to the n-terminus and c-terminus of the same helix and (assuming only short connections) between pairs of vertices which link the c-terminus of one helix with the n-terminus of the next.

Figure 3.6 explains the use of finite domain constraints in HFE: In (a) the use of finite domains in the representation of globular helical windings. `Nhelices` is an integer in the domain `{3,4,5,6}`. Structural constraints on windings are: (b) all vertices must be distinct (c) the position of the first helix is constrained to generate all and only rotationally distinct windings (d) and adjacency constraints.

```
(a)
set_up_FDs(Nhelices,Winding):-% (+integer,-ListFD).
        Vertices is 2*Nhelices,
        MaxVertex is Vertices-1,
        set_up_FDs(Winding,MaxVertexNo,Vertices).

set_up_FDs([],_,0):-!.
set_up_FDs([Vertex|RestVertices],MaxVertexNo,VertexCount):-
        Vertex: 0.MaxVertexNo,
        VertexCount2 is VertexCount-1,
        set_up_FDs(RestVertices,MaxVertexNo,VertexCount2).
(b)
apply_structural_constraints(N,ListFDVertices) :-
        alldistinct(ListFDVertices),
        fix_first_helix(N,ListFDVertices).
(c)
?- parallel fix_first_helix/3.
fix_first_helix(3,[0,1|_]).        fix_first_helix(5,[0,1|_]).
fix_first_helix(4,[0,1|_]).        fix_first_helix(5,[1,0|_]).
fix_first_helix(4,[0,3|_]).        fix_first_helix(5,[1,2|_]).
fix_first_helix(4,[3,0|_]).        fix_first_helix(5,[2,1|_]).
fix_first_helix(4,[2,3|_]).        fix_first_helix(5,[1,5|_]).
fix_first_helix(6,[0,1|_]).
(d)
find_windings_with_short_connections(_,[_]).
find_windings_with_short_connections(N,[V1,V2|T]):-
        adj(N,V1,V2),
        find_windings_with_short_connections(N,[V2|T]).

find_windings_with_any_connections(_,[]).
find_windings_with_any_connections(N,[V1,V2|T]):-
        adj(N,V1,V2),
        find_windings_with_any_connections(N,T).
```

Figure 3.6: **Use of finite domain constraints in HFE**

4 Parallelization Strategy

Parallelism in ECL^{i}PS^e gives the user access to parallel enumeration and search facilities which can be used to boost the efficiency of algorithms involved in optimization and search problems. The ability to employ parallelism in CLP applications is dependent upon non-determinacy in the application program. In the ECL^{i}PS^e language as with other logic programming languages such nondeterminacy is manifested in code for which more than one clause can match a given goal potentially producing backtracking on a sequential device. When parallelism is introduced into an application, however, an appropriate level of granularity must be chosen such that the overhead of initiating parallel processing is outweighed by the resulting increase in efficiency.

For CBS1e the effects of parallelism were investigated with a benchmark which finds all the solutions for a 10, 12 and 14 stranded sheet using constraints c1, c2, c3, c5 and c6 with strand 3 designated arbitrarily as unconserved giving 28, 45 and 66 solutions respectively from. In the case of the 10 stranded sheet the problem space is 4.75.1011

possible topologies. This was done using the predicates marked as parallel in Figure 3.2 and by replacing the definition of `instantiate/2` with a call to `par_member/2` in the code listed in Figure 3.1 using a Sequent Symmetry with up to 12 Intel 86386 processors.

The strategy for parallelization adopted with CBS1e and CBS2e was to simply identify obvious sources of indeterminacy in a serial ECL^{i}PS^e implementation and then as mentioned above, specific predicates were marked as parallel. The benchmarks were then used to investigate the effectiveness of the parallelization. At the time that this work was completed, a tool called PARTRACE, which illustrated the allocation of goals to different workers was used to visualize the effects of the parallelism.

It did not prove at all difficult to achieve creditable increases in performance using a simple parallelization strategy and no serious problems were encountered in debugging the parallel versions. The important factor, was however, choosing an appropriate benchmark to assess the value of the parallelization. If the benchmark does not provide a sufficiently large problem space, then the effects are not obvious. It is also important to use a well designed benchmarking methodology and compare results for returning all solutions.

4.1 Benchmarking methodology

For each of the three queries (10-strands, 12-strands and 14-strands), 3 times were recorded with each of 1, 2, 4, 8 and 12 processors (12 being the maximum on this particular machine).

The methodology for measurement of ECL^{i}PS^e program performance time evolved during the project until it settled upon an approach guided by the statistical principals of sampling theory. Essentially each goal is run number of times (approximately 10N, where N is the maximum number of workers) with the precise number of workers (processors) selected randomly for each run. Then each of the following time measures is computed: the elapsed time, the user cpu time, the system cpu time.

Ideally the sum of user cpu time and system cpu time should be approximately equal to the elapsed time, though the latter will include time for disk access, or time that other processes are occupying the cpu.

5 Performance Debugging and Optimization

In all measurements of speed-up, benchmarks were run on at least two different parallel computer systems; A SUN MicroSystems dual processor 630MP (Sparc II processors), a Sequent Symmetry (12 Intel 386 processors) or a 4 processor ICL DRS6000 (Sparc II processors). The speed-up results we obtained were all essentially the same, but there were of course differences in the absolute performance due to the different speeds of the individual processing elements.

Measurements made using CBS1e exhibit almost ideal behaviour (Figure 5.1). By increasing the number of strands in the sheet the problem space could be extended to show that as the problem space increases, the benefits of parallelism become close to linear with

respect to the number of processing elements. With such results, there is little scope for significant optimisation.

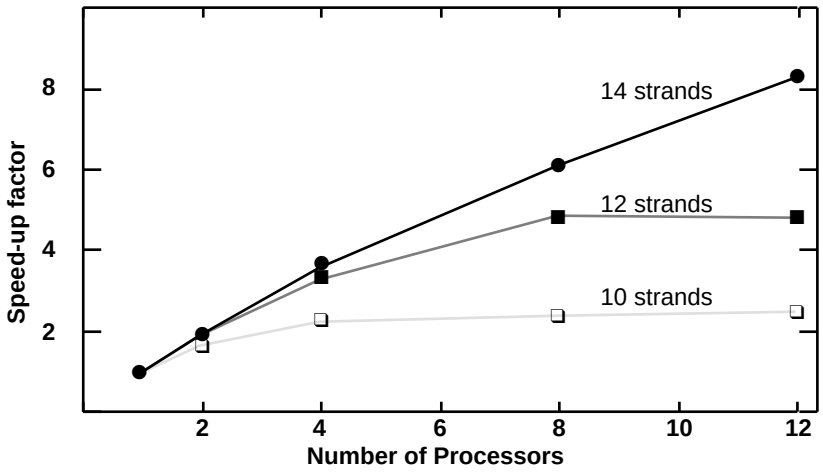


Figure 5.1: **The benefits of parallelism as a function of problem space in CBS1e.** The speed-up due to parallelism corresponding to three example β-sheets of increasing protein size (10, 12, 14 strands). All solutions were obtained and the elapsed time measured for up to 12 processors on a Sequent Symmetry.

Analysis of the performance of CBS2e and HFE however, illustrated two different ways in which program structure can affect the ability to exploit parallelism.

With CBS2e, the initial benchmarking used the same data as we had utilised with CBS1e. However, the initial data analysis (not shown) revealed a super-linear speed-up for 2 processors (18-fold) but no additional gain from the addition of further processors on the Sequent Symmetry. A possible explanation for the super-linear speed-up was that the problem space was small and the solutions unevenly distributed (in the search space seen by the second PE). Consequently, a further benchmark query was run employing an alternative hypothetical set of costs allocated to the CBS2e constraints. In this new cost table all constraints were given lower penalties for being false with the result that there were 450 minimum cost solutions each of cost of 24 taking approximately 5 minutes to run with one processor on the Sequent Symmetry.

An analysis of this query again showed good speed-up with the addition of the second processor (a speed-up factor of 2.14) but no additional performance enhancements with the addition of further processors. Indeed, the overall elapsed time got worse (Figure 5.2). This behaviour suggested that there was a potential bottleneck in the program and efforts were made to identify the source of the problem. After experimenting with the PARTRACE tool for visualizing the scheduling of goals to processing elements, and examining the structure of the program, a close coupling between subgoals responsible for cost minimization and solution generation was identified.

Creating a new version of CBS2e ensured that minimization was clearly separated from the solution generation (search and labelling) subgoals and led to an overall improvement in performance. By measuring the time taken in the different stages (minimization and generation) it is possible to see how the minimization procedure is interfering with other parts of the program performance. In Figure 5.3 execution time data were collected on the minimization and generation subgoals. The generation subgoal shows essentially

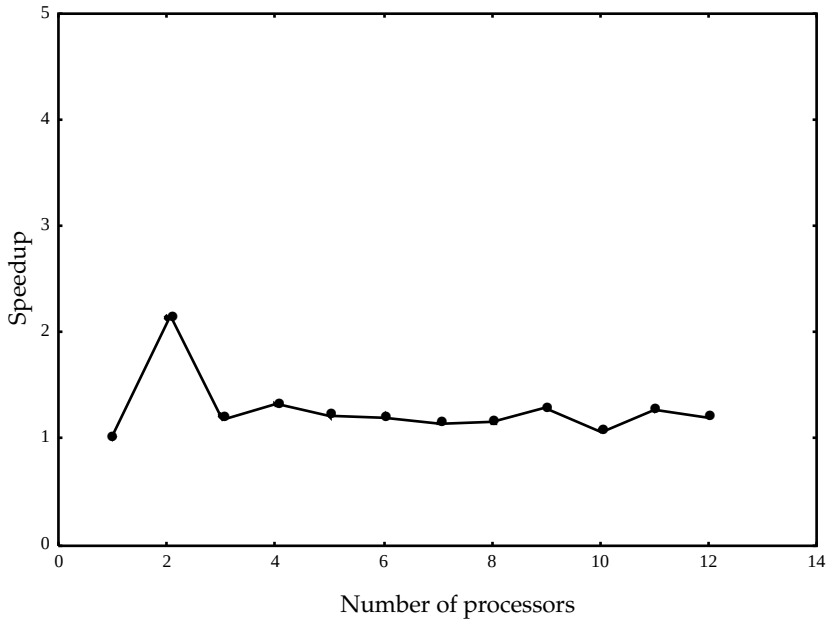


Figure 5.2: **The speed-up performance of CBS2e:**
Using a modified cost table for CBS2e, the benefits of parallelism were measured on a 12 processor Sequence Symmetry. The data shows that a bottleneck exists in the program structure.

ideal behaviour with near-linear speed-up, however, this is degraded by the asymptotic performance of the minimization subgoal. The overall performance is thus a compromise between the performance of the two predicates.

The helix folding program HFE is a much simpler program than CBS1e or CBS2e and it was thus possible to use it to evaluate whether introducing parallelism at two independent points provided truly additive benefits. The benchmark we used to test this with was the generation of all windings with short connections for 5 helices (1,205 solutions, from a problem space of 19,200 windings).

Benchmarks were run using ElipSys version 0.6.2 with the default stacks on a quiet Sequent Symmetry. Each benchmark was run on approximately 100 trials with the number of workers randomized on each trial.

In this experiment, the two predicates chosen for parallelism were `adj/3` and `fix_first_helix/2`. They were both non-deterministic and had five possible clause matches for 5-helical bundles. The difference between the predicates was that `fix_first_helix/2` is only ever called once, whereas `adj/3` is used as both a constraint and in the search and labelling phase of HFE. It would therefore be expected that `adj/3` has a greater potential for parallelization and this is demonstrated in Figure 5.4 where the curve shows an almost ideal speed-up pattern with a near-linear increase in performance with the number of processors. In panel b, the dependency between inherent parallelism and nondeterminacy is well illustrated with `fix_first_helix/2` as the only parallel predicate. Since this predicate is called only once, it follows that more than five processors should not lead to speed-up when this is the only parallel predicate. In fact the speed-up curve shows a strong linear trend with a gradient of about 0.8 up to five processors (Figure 5.5) but then is perfectly level. This shows that `fix_first_helix/2` is a use-

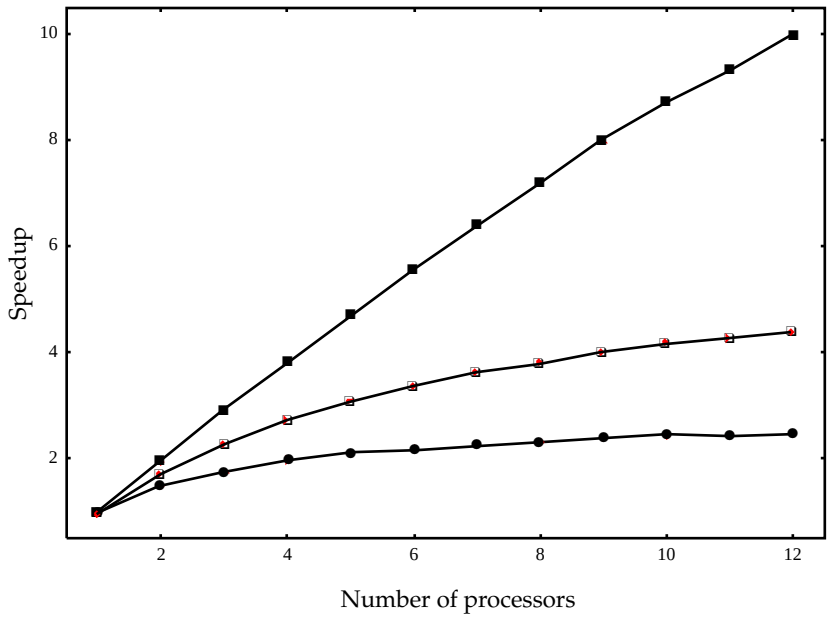


Figure 5.3: **The speed-up performance of a modified CBS2e:** CBS2e was modified to separate out the cost minimization and solution generation parts of the program and the contribution of the different aspects factored out in the measurement of speed-up. The contribution of the minimization phase (solid circles) effectively limits the overall speed-up (open squares) although the solution generation exhibits good performance characteristics (solid squares).

ful predicate to declare as parallel in isolation with up to five processors. When other predicates are parallel no such limitation exists of course. Figure 5.6 shows the speed-up curve with both `adj/3` and `fix_first_helix/2` parallel. This gives the best speed- up figures of all for this benchmark (a median factor of about 9 for 12 processors), illustrating that some combinations of parallel predicates do lead to essentially additive levels of parallelism.

6 Conclusions

The work described in this chapter has reviewed the way in which we have exploited parallel CLP in applications of protein topology prediction. Our initial motives for using parallel CLP were that it would provide a technology for building CLP applications that would scale well when applied to real scientific problems. In the domain of protein topology prediction, this was indeed the case. The parallelization of CBS1e enabled us to consider extensions to encompass uncertainty in the definition of protein folding rules without significant loss of performance. This was a significant advantage from a scientific point of view and we are now in a position to exploit these programs and make them available through a graphical interface and a World Wide Web service.

One of the reasons that the application of parallel CLP to protein topology prediction was that we had a rich set of constraints which applied to both local and global aspects of the problem space. Since, however, the problem space was not fully constrained by these constraints, the solutions must eventually be found by search which could be executed

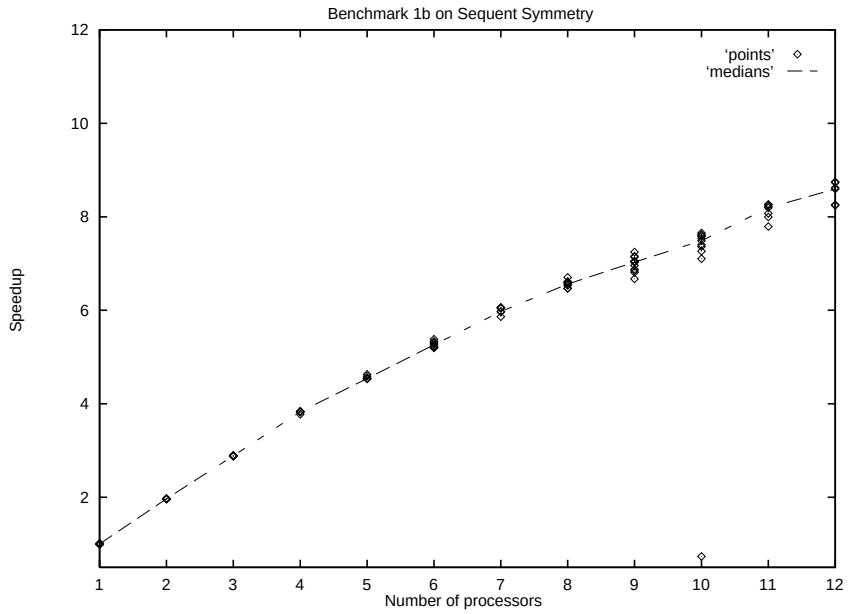


Figure 5.4: **The speed-up performance for HFE with `adj/3` only set parallel.**

very effectively in parallel. In a second application domain, the assembly of large scale physical genetic maps of human chromosomes [CRD94] we observed similar benefits from parallelism (i.e. performance increased essentially linearly with respect to the number of processors used).

Excluding the development of user interfaces, the major effort in all these applications was the development of the appropriate constraint representation. The initial introduction of parallelism was largely straightforward. However, In the case of CBS2e and HFE, more effort was needed to quantitatively analyse both interference (CBS2e) and complimentarity (HFE) between parts of our programs. These issues were identified by a combination of performance profiling, a knowledge of the program structure and empirical analysis based on a sound benchmarking methodology. Our experience with parallelism is that the potential arises directly from non-determinacy in all our ElipSys/ ECL^{i}PS^e applications, so that a modest amount of informal experimentation can lead to some speed-up. This process can then be substantially extended by the use of parallel performance debugging tools and procedures.

The next generation of performance debugging tools should address the issues of providing a wide view of program performance, particularly with a view to rapidly identifying serial bottlenecks and interference between predicates (probably because of inappropriate use of parallelism at too fine a granularity).

A major advantage for the molecular biology applications has been the ease of moving between different hardware platforms (SUN, ICL, Sequent).

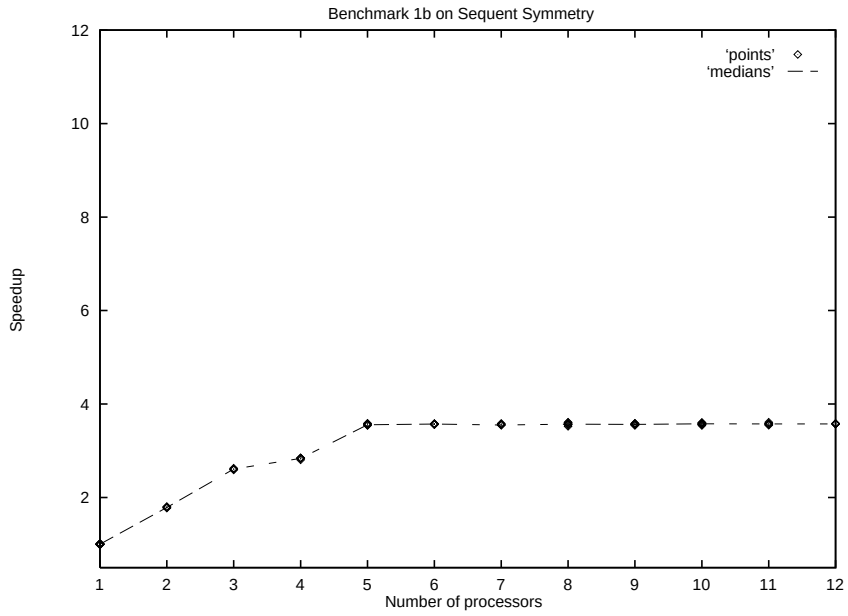


Figure 5.5: **The speed-up performance for HFE with `fix_first_helix/2` only set parallel**

Chapter 7.

A Tourist Advisory System for Greece

Panagiotis Stamatopoulos and Isambo Karali

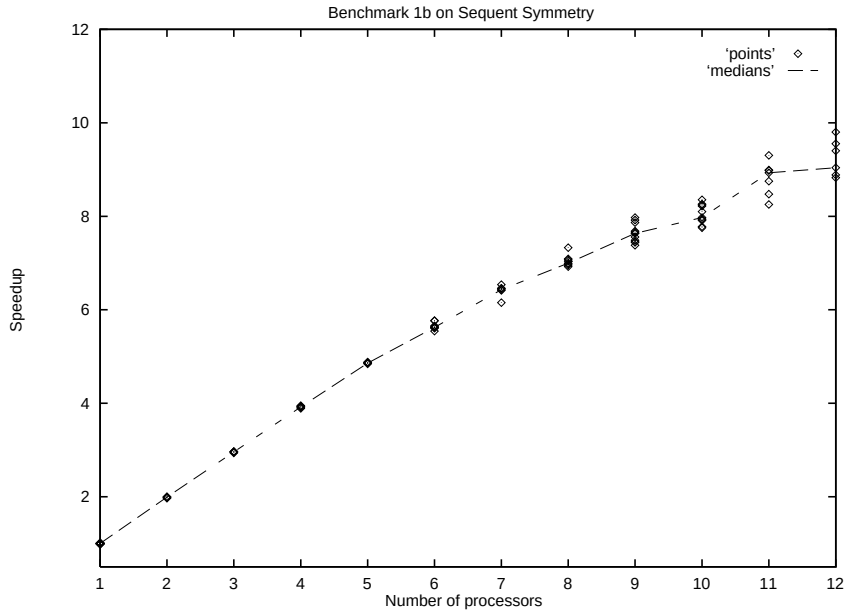


Figure 5.6: **The speed-up performance for HFE with both `adj/3` and `fix_first_helix/2` set parallel.**

1 Introduction

MaTourA [HSK+94] is an ECL^{i}PS^e [ECL95] application from the area of tourism, developed by the University of Athens and Expert Systems International S.A. in the context of the ESPRIT project APPLAUSE [LRS+93b, LRS+93a]. The purpose of MaTourA, which is a **M**ulti-**a**gent **Tour**ist **A**dvisor about Greece, is to leverage the services offered by travel agencies by providing an interactive way to construct personalized tours and handle the complex underlying tourist information. Given the preferences of a tourist for his/her holidays in Greece, MaTourA produces a personalized tourist plan, taking into account the constraints of the tourist and information about potential visits, upcoming events, particular sites, accommodation and transportation.

MaTourA demonstrates the different technologies integrated in the ECL^{i}PS^e system in one application, i.e. parallel CLP, the embedded database functionalities and the WWW server support, and couples them with the technology of multi-agent systems. The latter is a major research area of Distributed Artificial Intelligence (DAI) [Huh87, BG88]. In a multi-agent system, the idea is to have agents of various types and capabilities to cooperate in problem solving. The cooperation among the agents is achieved in various ways, for example by using a blackboard [HR85, EM88], contract net [Smi80, DS83] or actor [AH85, Hew88] model. In the MaTourA case, a blackboard approach has been adopted.

2 MaTourA Architecture

The MaTourA system comprises a set of autonomous agents reflecting the procedures involved in a tourist advisory environment. These agents are:

Tour Generation Agent: It is responsible for the construction of personalized tours, taking into account user wishes. These tours are time/location schedules providing, for each day, a specific timetable. This agent is the most computationally intensive among the MaTourA agents. It is the one where the parallel CLP technology is exploited in the MaTourA system. This chapter deals with the problem faced by the Tour Generation Agent and how it is tackled in ECL^{i}PS^e.

Activity Agent: It holds information about activities and is able to answer requests coming from other MaTourA agents. Activities are possible tourist's visits to various spots, such as museums, galleries, archaeological sites, beaches etc.

Event Agent: It holds information about events and, as is the case with the Activity Agent, it answers requests relevant to this information. Events differ from activities in that they are short term shows with rather temporary nature, such as exhibitions, music concerts, theatre performances etc., while activities are more permanent in time.

Site Agent: This agent deals with the sites of Greece, i.e. the geographical entities of the country, such as villages, cities, islands, regions etc., which are organized in a site inclusion relation. The concept of sites is an important one for MaTourA, since the whole tour generation facility works considering the sites as a fundamental starting point.

Accommodation Agent: It handles information about lodgings where a tourist may be accommodated and supplies it after the appropriate requests are made.

Transportation Agent: Information about connections between sites via different transportation means is managed by this agent. All possible transportation means are covered, that is private car, bus, train, boat and airplane. Besides the support of plain information retrieval, this agent is capable to solve various routing problems.

Ticketing Agent: This agent holds information about connections which are established between different sites all over Greece by public transportation means. Information relevant to prices, timetables, facilities etc., is maintained.

Package Tour Agent: It is responsible for handling package tour information. A package tour is a precompiled tour, as it has been constructed by a travel agency.

User Interface Agent: This agent controls all user interaction with the MaTourA system. It acts as an intelligent front-end to the functionality provided by the other MaTourA agents.

The MaTourA agents, except the User Interface Agent, have a common property. They all accept requests for processing, one at a time, do some computation for their resolution and, then, send back the results. For this reason, these agents are called "computation agents". The requests sent to the computation agents are formulated as messages following a formal Prolog term syntax. Moreover, the computation agents, except the Tour Generation Agent, perform direct processing of raw data contained in their local databases. Actually, these are information servers and they are called "database handling agents". The database handling agents exploit the BANG file functionality provided by ECL^{i}PS^e.

Earlier versions of the MaTourA system were implemented in the predecessor of ECL^{i}PS^e as far as parallelism is concerned, the ElipSys language [BBDR+90, Eli93]. Now, the whole application has been ported to ECL^{i}PS^e, so as to profit the most from the language's advanced features.

As far as the interaction among the MaTourA agents is concerned, a three-layered communication framework has been developed for this reason [SMH94]. This framework is a library of ECL^{i}PS^e (formerly of ElipSys) which, at the higher level, provides a set of point-to-point communication predicates for message exchange between two agents. This layer is based on a Linda-like blackboard architecture [CG90] which, in turn, is supported, at the lower level, by a set of primitives for handling stream sockets in the Internet domain [Ste90]. The experience from the implementation of the MaTourA system is that the structuring principle of multi-agent systems, as this is supported by the developed communication framework, enhances the horizon of ECL^{i}PS^e to directions where large scale development and distributed computing are central issues.

Besides the structuring of the MaTourA system as a set of cooperating high-level agents, the concept of subagents has been introduced as well. A subagent may be viewed as an entity which carries out one of the subtasks that a high-level agent has to accomplish. However, while the high-level agents are, more or less, complete processing elements, which might be also spatially distributed, the subagents of a high-level agent are tightly coupled problem solvers that share a common computing environment and exchange information through logical variables, rather than using network facilities.

The rest of this chapter is devoted exclusively to the Tour Generation Agent. The reason is that the other MaTourA agents are rather simple "computing machines" where there is no need for exploitation of parallel CLP. On the other hand, the Tour Generation Agent is the most computationally intensive agent of MaTourA, since it has to solve an extremely hard combinatorial problem. The focus of the following discussion will be on this problem and the way it is tackled in the parallel CLP environment of ECL^{i}PS^e. Other interesting issues, such as the communication framework for the high-level agents, the concept of subagents, the functionality and the structure of the other agents etc., are presented elsewhere [PA92, HSP+93, HSM+93, HKS+94, XSG+94, HSKG95].

3 Problem Description

In a tourist advisory environment, a common problem is to construct tours, that is sequences of visits to various places, spots etc., which accommodate the preferences of individual tourists. In the MaTourA system, this functionality is provided by the Tour Generation Agent (TGA).

As it is the case of every MaTourA agent, the TGA accepts requests which express specific user requirements. There are four types of such requests whose main characteristics are given in the following in increasing degree of complexity:

1. A full time/location schedule is provided to the agent, e.g. 15 Jul 95 — 19 Jul 95 in Athens, 19 Jul 95 — 25 Jul 95 in Rhodes, 25 Jul 95 — 31 Jul 95 in Heraklion.

2. The visit period of the tour is given, but not the very locations. However, a wider area is supplied or, alternatively, a starting location for the tour, e.g. 15 Jul 95 — 5 Aug 95 in Cyclades.

3. Specific locations are given but no information about time, e.g. a tour in Thessaloniki, Edessa, Kavala, Alexandroupoli.

4. Neither specific locations nor information about time is provided, but a tour with a given starting location or in a wider area is requested, e.g. a tour starting from Ioannina.

For each type of request, cost criteria as well as interest preferences are given to the agent. Moreover, acceptable transportation means and accommodation constraints are taken into consideration. Finally, a set of Daily Plan Templates (DPTs) are also supplied. Each DPT corresponds to a typical day that the tourist would like to spend on a specific location. A DPT consists of time period/action pairs, e.g. 10:00 — 14:00 swimming, 18:00 — 21:00 sightseeing.

Although the requests of type 1 are the simplest ones, they are quite complex and difficult to cope with. For the requests of other types, heuristic procedures have been developed which transform such requests to requests of type 1. These procedures do not exploit parallel CLP, thus their presentation is not relevant to the current context.

A scenario is that after an interaction between the user and the system, through the User Interface Agent, the latter constructs a specific request and sends it to the TGA. Then, the result is computed by this agent and sent back to the User Interface Agent, so as to be presented to the user in a friendly way. As far as performance is concerned, the requirement is that a request should be answered in no more than 3–4 minutes, since MaTourA is an interactive system and, thus, it is not meant to be used in some kind of batch mode.

A typical (type 1) request to the TGA is the following:

```
tourgen_req1([culture, history],
        100000,
        accom([hotel], 'C'),
        [flight],
        [local_tour(from_to(5/8/95, 9/8/95),
                'Athens',
                 [template([time_act(10:00, 13:00, [sightseeing]),
                            time_act(15:00, 16:00, [learn])])]),
              local_tour(from_to(9/8/95, 12/8/95),
                 'Heraklion',
                 [template([time_act(13:00, 14:00, [sightseeing])]),
                  template([time_act(16:00, 18:00, [learn])])])],
        no_opt/10)
```

In this request, the TGA is asked to construct a tour in two sites, i.e. in Athens from 5/8/95 to 9/8/95 and, then, in Heraklion from 9/8/95 to 12/8/95. For Athens, one DPT

is provided which states that, for each day, the tourist wants to do some sightseeing from 10:00 to 13:00 and to learn things from 15:00 to 16:00. For Heraklion, there are two alternative DPTs for every day, one requesting sightseeing from 13:00 to 14:00 and another for learning from 16:00 to 18:00. The terms "sightseeing" and "learn" are actions which have to be analyzed to activities and events of specific categories that may fall under these actions. In addition, in this request, it is stated that the tour should contain activities and events that present "culture" and "history" interests, the cost of the tour should be around 100,000 drachmas, the accommodation should be at hotels of at least "C" class and the transportation should be by air. Finally, a maximum number of 10 tours is requested which have to be computed without using any optimization facilities.

In the following, since requests of types 2, 3 and 4 are mapped to type 1 requests, only the processing of the latter is discussed. Moreover, this processing involves some "satellite" tasks, such as:

- distributing the available budget among accommodation, transportation and visits
- preprocessing of requests for lengthy tours to allow replication of visits to the same activities and events
- expanding the interest macros to basic interests
- communicating with the Transportation and the Accommodation Agents to request the satisfaction of the corresponding constraints stated by the user
- communicating with the Site Agent to get the site keys of the involved sites

These tasks do not fall into the scope of the presentation of this chapter, since they do not exploit parallel CLP as well, thus, they will not be discussed any further.

So, the problem that will be considered from now on is the filling of DPTs, for every site and every day, with specific activities and events which are qualified for this purpose, that is they satisfy the stated constraints. Actually, this is the computationally intensive combinatorial problem faced by the whole application. Dealing with this problem is the most time consuming part of the MaTourA system.

Referring again to the previously presented example, the main input to the problem is the fifth argument of `tourgen_req1/6`, which is a list of "local tour" structures. Each structure of this kind corresponds to a specific site and a specific time period to be spent by the tourist in this site. For each site, a list of alternative DPTs is given to use them for the construction of the day schedules. Such schedules are not constructed for the transportation days, i.e. the day that the tourist arrives at the first site, the days of travelling from one site to another and the day the tourist leaves from the last site. In this example, it is requested to create tours for the dates 6/8/95, 7/8/95 and 8/8/95 (in Athens) and the dates 10/8/95 and 11/8/95 (in Heraklion). One of the tours that have been computed as solutions to this request is:

```
6/8/95  ---  Athens
    10:00 - 13:00 --> Acropolis
```

```
    15:00 - 16:00 --> Keramikos
7/8/95  ---  Athens
    10:00 - 13:00 --> Acropolis Museum
    15:00 - 16:00 --> Gennadios Library Collections
8/8/95  ---  Athens
    10:00 - 11:30 --> Physics Collection of Evgenidis Institute
    11:30 - 13:00 --> Ancient Market
    15:00 - 16:00 --> Ag. Apostoloi
10/8/95  ---  Heraklion
    13:00 - 14:00 --> Koules
11/8/95  ---  Heraklion
    13:00 - 14:00 --> Venetian Wall
```

In the following, the Tour Construction Problem (TCP) just presented will be considered and analyzed from various points of view and the way the parallel CLP technology of ECL^{i}PS^e is used to deal with it will be presented in detail.

4 Characterization

A major characteristic of the TCP which has strongly influenced the adopted approach is the very dense solution space of the problem. It is not possible, and, certainly, not meaningful, to compute all, in the mathematical sense, solutions to a given request. The reason is that there is a large number of such solutions, among which there exist tours which are almost identical except of a single activity or event. In addition, the almost independent subproblems for each site imply the computation of the Cartesian product of the sets of solutions for each site to create the set of global solutions. This is another source of the huge number of solutions. What is needed, then, is to present to the user a reasonable set of sufficiently different tours, so as to give him/her the opportunity to make the final decision. This concept of "sufficiently different tours" is modelled appropriately and incorporated into the system. Of course, the user may ask for a single solution, which makes the whole computation simpler.

As far as searching for an optimum solution is concerned, unfortunately, the raw data provided by the Activity and Event Agents do not distinguish between "good" and "bad" activities and events. However, an interest attribute has been introduced virtually with the aim to quantify the quality of a potential visit and, thus, to contribute accordingly to the tour it participates. This interest has been defined as a heuristic function of the duration of a visit and the number of different interest types it presents. With this formulation, an optimization TCP is considered as well.

Another issue that has to be discussed is whether the nature and the size of the system databases have affected the design of the system. Even from the very beginning of the development of MaTourA, databases with real data have been used, though incomplete and not from every region of Greece. There has not been any attempt to start with some kind of randomly created data, since this might result to wrong design decisions. However, the initial data have not been sizeable enough and, thus, the developed prototype

has been proved inadequate later on, due to the simplicity of the encountered methods. The prototype implementation has been radically changed since then, and, now, it may be said that the system performs really well. In addition, the databases, at their current state, contain the full data from Athens and the Crete island and it is foreseen that in the very near future the full data from the Peloponnese region will be available. The incorporation of data from other regions will add only to the functionality of MaTourA without increasing the complexity of the problems that have to be solved. This is due to the geographic modularity which is reflected both in the design of the system and the provided functionality for tour construction.

The TCP, as it was presented in the previous paragraph, has not been tackled by any method in the past. A similar, though much simpler, problem to this was faced in the PETINA application [HSKM92, SKH92, SKH93], which was developed by the University of Athens in the context of the EDS project [SHL+92]. PETINA was implemented in the ElipSys language and it was the first attempt to employ the parallel CLP technology in solving combinatorial problems from the area of tourism. The results were quite satisfactory, both as far as the exploitation of constraints and the performance gain on parallel platforms are concerned. To this direction, it was decided to tackle the TCP of MaTourA using the same technology. The reason is that, apart from the experience gained from the development of PETINA, there was much evidence that the combinatorial nature of the TCP would lead to a significant profit from parallel CLP. The whole TCP might be faced as a very large constraint satisfaction problem exploiting parallelism at the labeling phase or it might be broken down into smaller constraint satisfaction subproblems which could be solved in parallel and have their results combined afterwards. The first approach would certainly require the employment of very efficient labeling strategies, in order to be reasonable. Thus, it was decided to follow the second approach and the results, as it will be shown next, are very good.

5 Constraint Modelling and Prototyping

From a theoretical point of view, the TCP is a huge constraint satisfaction problem. It might be modelled as such, but it is doubtful whether a global formulation like that would lead to the desirable results. For this reason, it has been decided to divide the problem into subproblems and deal with them, instead. As it will be explained in the following, this division facilitates also the exploitation of parallelism.

At the first level of problem partitioning, a tour is considered as a sequence of subtours, each one corresponding to a specific site. At the next level, each site subtour is a sequence of day subtours, one for each day that is being spent to the given site. More on this decomposition of the original TCP will be said in the next paragraph, where parallelism issues are discussed.

So, what has to be solved now is the generation of a given number, say N, of sufficiently different subtours for a specific day at a specific site. These subtours have to be instantiated DPTs from the available ones for the given site. This instantiation has to be done with activities and events taken from a pool that has been created for this reason. The pool contains candidate visits for every possible time period of each template for all days in all sites of the original request. These visits have been chosen so as to satisfy the

interest constraints of the request and to belong to an activity or event category that corresponds to the action(s) of the time period into consideration. This pool is created at a pre-processing phase by sending appropriate requests to the Activity and Event Agents. The only thing that has to be checked when putting a visit into a DPT is the time periods when it is active. Moreover, the duration attribute of the visit has to be respected.

The construction of the N sufficiently different subtours is carried out by a recursive procedure which, at each iteration, computes a new subtour which has to be a filled DPT from the available alternative ones. In addition, this subtour has to be not very similar to any other already constructed subtour. These properties of a subtour are expressed in terms of constraints on finite domains in a way that is presented below. One thing that has to be mentioned also is that it is not guaranteed that it is possible to find exactly N sufficiently different subtours. Actually, N acts as an upper limit for the number of subtours to be computed.

A day subtour, as it has been already said, corresponds to a DPT. Thus, a DPT is chosen non-deterministically and for each time period in this DPT it is decided how many visits will be inserted there. This number of visits is computed heuristically from a formula that involves the duration of the time period to be filled and the average duration of candidate visits. Actually, not only the result M of this formula is considered, but $M+1$ and $M-1$ in a non-deterministic way as well. Thus, there is a reasonable variety in the number of visits in each time period of the DPT. However, in order to reduce the complexity of the problem, if more than 5 visits have to be inserted into a given time period, this period is split into shorter ones where less than or equal to 5 visits are put. This is equivalent to fixing the time of the transition between two visits in the original problem. Now, having decided on the number of visits, say K ($= M$ or $M+1$ or $M-1$), in a time period, 3 domain variables are defined for each one of the K visits (a total of $3 \cdot K$ variables), one for the starting time of the visit, one for the ending time of the visit and one that represents an index to the pool of candidate visits. The duration, the opening time and the closing time of a visit are also domain variables which are related to the index and, thus, to each other through `element/3` constraints. The other constraints which are put are:

- The starting time of the first visit has to be equal to the starting time of the time period that is being filled.
- The starting time of the other visits have to be equal to the ending times of the previous visits.
- The ending time of the last visit has to be equal to the ending time of the time period that is being filled.
- Each visit has to be put in a time period, say (t_1, t_2), when it is really active, that is t_1 has to be greater than or equal to the opening time of the visit and t_2 has to be less than or equal to the closing time of the visit.
- For each visit, its duration has to be respected, that is it has to be equal to $t_2 - t_1$.

In order to make things more clear, the following are in ECL^{i}PS^e syntax the constraints which are stated for a single visit.

```
..........
element(Ind, Durs, Dur),
element(Ind, OTimes, OTime),
element(Ind, CTimes, CTime),
STime #= PrevETime,
ETime #= NextSTime,
STime #>= OTime,
ETime #<= CTime,
Dur #= ETime - STime,
..........
```

`STime` and `ETime` are the starting and ending times of the visit respectively (ranging over `0..1440`) and `Ind` is the index to the pool of candidate visits that points to this visit. `Ind` ranges over a list of indices to legal visits, which are the ones that satisfy the interest preferences of the tourist and the action requirements for the time period into consideration. `Durs`, `OTimes` and `CTimes` are the lists of the durations, opening times and closing times respectively of all candidate visits. Thus, `Dur`, `OTime` and `CTime` represent the duration, opening time and closing time respectively of the visit to which `Ind` points to. Finally, `PrevETime` is the ending time of the previous visit and `NextSTime` is the starting time of the next visit.

The resulting subtours for each time period in a DPT are concatenated to form the day subtour. At the day subtour level, another set of constraints which is stated is that all visits have to be different to each other. Finally, each day subtour has to be sufficiently different from the already computed ones. The latter constraint is expressed in the following way. If a day subtour contains L visits, L random numbers are generated in the range $[1, L]$. For each random number R, the R-th visit of the day subtour is stated to be different from all visits in the day subtours computed so far. This results to having a reasonable number of visits of a subtour different from the visits in the previous subtours, which implements the idea of "sufficiently different tours". As far as implementation is concerned, if for two visits, that correspond to indices `Ind1` and `Ind2` to the pool of candidate visits, it has to be stated that they are different, the following constraints are set.

```
..........
element(Ind1, Ids, Id1),
element(Ind2, Ids, Id2),
Id1 ## Id2,
..........
```

`Ids` is the list of identifiers of all candidate visits and, thus, `Id1` and `Id2` are the identifiers of the two visits into consideration.

The total virtual interest of the visits in a day subtour may act as a cost function which has to be maximized. This is an option of the system which, with the overhead of a more heavy computation, results to "good" tours. The problem is that the randomness which has been introduced in the selection of visits for stating the "sufficiently different tours" requirement and the decomposition of the original TCP into smaller subproblems, do not lead necessarily to the N most interesting subtours. It may be the case that the

one computed at the i-th iteration is less interesting than that of the $(i+1)$-th iteration. Thus, the optimization feature supported by MaTourA is oriented more to the assessment of the technology rather than to the provision of a useful functionality for the end-user.

The formulation of the problem, as it is presented in this paragraph, has been adopted after a long way of experimentations and development of prototype versions. From the very beginning of the design of MaTourA, the aim has been to couple the user appreciation, as far as the tour generation facility is concerned, with the required efficiency. Many attempts have been done with different formulations in the direction of the exploitation of other facilities of the employed language platforms. Even the `contigs/5` constraint has been used, during the ElipSys era of MaTourA, but without having the expected benefits. In order not to commit to a specific number of visits in a given time period, various formulations had been employed with the disadvantage of having a huge number of domain variables and constraints on them. For example, in such a formulation, the number of the defined domain variables is proportional to the total tour time, in minutes. For the example given in this chapter, this is something like 1000, while with the adopted approach, the number of introduced domain variables is proportional to the number of visits, i.e. around 10. This, in conjunction with the powerful constraint mechanism provided by ECL^{i}PS^e has lead to significant performance gains.

6 Parallelization

As far as parallelism is concerned, this facility has been exploited in various points of the solution of the TCP. Since MaTourA is a new application, parallelism has been considered from the beginning of the development and, thus, the well-known problems of parallelizing existing sequential programs have not been faced.

One major source of parallelism is related to the concept of subagents which have been implemented through the introduced generalized AND-parallelism (gAND-parallelism) construct (`&/1`). The gAND-parallelism is a variation of the normal AND-parallelism of ECL^{i}PS^e (`&/2` predicate) which is based on a data-parallel like execution of more than two goals. These ideas have been applied successfully in two cases, where independent subagents are employed:

- Global tours, that is the required solutions, are computed by generating a set of site subtour subagents which work in parallel for the construction of subtours for each site. The subproblems that the subagents have to solve are not really independent, since the cost of the global tour, which is the sum of the costs of the subtours, has to be less than a certain limit. However, these subproblems are made independent by distributing a priori the available budget to the sites proportionally to the time periods the tourist is going to stay at each site. Now, if N tours are required, each site subtour subagent is requested to compute N subtours for each site. The actual number of computed subtours may be less than or equal to N for each site. Then, at a post-processing phase, an algorithm is applied, which constructs M global tours, where M is the maximum number of subtours computed for a site by a site subtour subagent ($M \leq N$). This combination of subtours to construct tours is computationally cheap.

- A number of site subtours is computed by assigning to especially created for this reason day subtour subagents, one for each day to be spent in the specific site, the construction of the same number of day subtours. That is, if N site subtours are required, each day subtour subagent is requested to create N day subtours. In this case also, it is not guaranteed that the day subtour subagents will be able to compute exactly N subtours each. The subproblems faced by the day subtour subagents are highly interdependent, since it is not only the cost issue that has to be considered again, but it is not desirable to propose to the tourist to visit the same activities and events more than once. However, it has been decided to let the day subtour subagents work independently in parallel and then, at a post-processing phase, take care of these limitations for the construction of site subtours. In this phase, it is tried to combine day subtours for every day in a site which contain completely different visits, in order to formulate a site subtour. This is repeated until it is not possible to create a new site subtour. The number of site subtours that may be created cannot exceed the minimum number of day subtours that have been produced for every day, which is, certainly, less than or equal to N. This post-processing phase may be proved time consuming in some cases, but experiments have shown that, on the average, the adopted approach is relatively efficient.

There are two more points where exploitation of parallelism has been introduced and tested. One is the data-parallel processing of alternative DPTs for the construction of one day subtour. The other is the data-parallel consideration of the three different numbers of visits in one time period of a DPT (M, $M+1$ or $M-1$ of the previous paragraph).

As a last comment on the parallelization issues, it has to be mentioned that it seems that the additional effort that someone has to put on the simultaneous consideration of parallel execution when developing from scratch an ECL^{i}PS^e application is significantly less than the performance benefits obtained from using this facility. What is more is that it is not always the case that difficult debugging may be needed due to the introduction of parallelism. Actually, in the MaTourA case, parallelism helped to locate a serious "design" bug that existed in the employed communication framework for the support of the development of multi-agent systems.

7 Performance Debugging and Optimization

The MaTourA system, more precisely its TGA, has been evolving during the last two years continuously. While the prototype versions have employed methods which have been rather simplistic, the current state of the application is satisfactory, since the functionality and performance targets have been achieved.

The most serious and most profitable sequential optimization that has been carried out since the beginning of the development of the TGA has been the minimization of the communication transactions between this agent and the other MaTourA agents, i.e. the Activity, Event, Site, Transportation and Accommodation Agents. These transactions have been proved very inefficient and have been a real bottleneck of the TGA. For example, at the initial versions of the system, a highly declarative formulation of the TCP has led to an unnecessary multiplicity of the requests sent to the Activity and Event Agents, actually

through backtracking. This, in conjunction with a trivially straightforward modelling of the involved constraint satisfaction problem, has given unacceptable performance results even with small subsets of the required raw data.

Much of the effort to improve the sequential performance has been put on the investigation of the efficiency of various formulations of the TCP. Many alternatives have been tested before committing to the approach that has been finally adopted. These alternatives include user-defined constraints, 0-1 formulations, disjunctive constraints, generalized propagation etc.

As far as parallel optimizations are concerned, no much work has been done related to this topic. The reason is that from the beginning of the design and the implementation, the parallel world has been the aimed execution environment and, thus, the whole development has been carried out having this in mind. There have been done a few experiments related to whether parallelism might be profitable in specific points of the adopted method, but this cannot be considered as real work on what is called "parallel programming". Taking into account the really good behavior of the TGA on parallel platforms, as it will be seen later, the credit has to be given to the way the ECL^{i}PS^e language supports the concept of parallelism.

This paragraph concludes with some performance results for the TCP as it tackled by the TGA of the MaTourA system. Two parallel platforms have been used, a Sun Sparc Server 1000 at the University of Athens and an ICL DRS 6000 at ECRC. The results refer to 14 representative non-optimization requests given to the TGA and the elapsed times correspond to the processing of the TCP. On the Sun platform, for each request, 4 runs have been measured with 1 worker and 12 runs with 2 workers. On the ICL platform, there have been measured 2 runs with 1 worker, 6 runs with 2 workers and 6 runs with 3 workers, for each request as well.

For each platform, the speedups S and the quasi standard deviations D are also given, as they are computed via the method proposed in [Pre94b], which is summarized in the formulas

$$S = \sqrt[mn]{\prod_{i=1}^{n}\prod_{j=1}^{m}\frac{T_s^i}{T_p^j}}$$

and

$$D = e^{\sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m}\frac{1}{mn}\left(ln\frac{T_s^i}{T_p^j}\right)^2 \;-\; \left(\sum_{i=1}^{n}\sum_{j=1}^{m}\frac{1}{mn}ln\frac{T_s^i}{T_p^j}\right)^2}}$$

The results are presented in the Tables 7.1, 7.2 and 7.3. As it may be seen, there are quite satisfactory speedups for most requests. This is true mainly for cases where there is potential for exploitation of parallelism, which comes from large numbers of sites to be visited, days to be spent at each site and alternative DPTs to be filled for each day. In case these characteristics are not met, e.g. in "req10" and "req13", the worker communication overhead may result in no additional gain when increasing the number of workers (compare for "req10" and "req13" the speedups with 2 and 3 workers on the DRS 6000

platform). However, for really computationally intensive requests, there may be considerable exploitation of parallel machines.

	1 worker		2 workers					
req01	6112	6491	3256	3235	3420	3760	3245	3159
	5582	6013	3239	3294	3879	3589	3420	4098
req02	2794	2805	2198	2294	2320	2328	2142	2323
	2671	2945	2332	2194	3371	2389	2473	2227
req03	2901	2802	1759	1812	1840	1807	1814	1981
	2805	3311	2048	1882	1893	2360	1945	2072
req04	4715	4632	3115	3122	3039	3210	2754	3115
	4084	5062	3321	3202	2957	3457	2771	3252
req05	18176	18453	11354	10891	11893	11756	10537	11820
	17247	18669	12677	11219	10581	11266	10590	11919
req06	13730	13065	8837	9845	9568	9342	8934	9712
	13013	13710	9485	6850	8489	9425	8615	8534
req07	52357	51935	26489	28329	28976	29078	25858	30006
	50673	53418	29421	28495	27889	27963	27860	27603
req08	3528	3619	2254	2376	2144	2474	2318	2226
	3588	3935	2531	2145	2147	2361	2114	2140
req09	5650	5489	2357	4784	2244	2343	2253	2237
	5188	5900	2285	2177	2705	3943	2181	4211
req10	6273	6353	3412	3450	4444	3475	3469	3530
	6055	6815	3555	3557	3881	3891	3477	3363
req11	16485	17126	10093	9371	9827	10339	9854	9858
	16073	17122	10449	9239	10521	10203	9367	10306
req12	17659	18316	9766	10118	11122	10346	10183	10008
	17223	19206	11008	10163	10499	11733	11145	10554
req13	7351	7419	3765	3299	4135	4101	3651	3653
	6482	7485	4213	3126	4503	4214	4032	3858
req14	14058	15070	8058	8094	7785	8144	7371	7712
	13886	15580	8590	8061	8185	8861	8062	9075

Table 7.1: Elapsed times (ms) on a Sun Sparc Server 1000

8 Conclusions

In this chapter, the computationally intensive problem, namely the TCP, faced by the TGA of the MaTourA system has been discussed and the way this problem has been tackled using the parallel CLP technology of ECL^{i}PS^e has been presented. It has been proved that a parallel CLP environment may help to master combinatorial problems from the area of tourism. In addition, if this technology is combined with the structuring principle of the multi-agent systems, the "programming in the large" concept is supported in an elegant, structured and efficient way.

	1 worker	2 workers			3 workers		
req01	20170	10820	9840	10070	10910	9130	10910
	20340	10950	10160	11460	11340	10220	9560
req02	8910	6970	7190	7250	7570	7390	7350
	8880	8620	8690	7540	7530	7430	7220
req03	9350	6100	6040	6000	4150	4170	4210
	9390	6040	6040	6040	4190	4170	4180
req04	13410	9200	9140	9260	6730	6290	6900
	13450	9220	9280	9220	6760	7290	7230
req05	56290	38370	38500	37910	28100	27120	24320
	56550	38470	38560	38330	27430	28910	26500
req06	44500	22500	22640	28530	22030	22310	22500
	44530	29680	30620	22600	23100	22840	22170
req07	168290	92600	92350	86290	65330	63880	63930
	168500	90580	92850	92280	63860	64440	65620
req08	11630	6790	6850	6910	5800	5780	6200
	11640	6920	6950	6760	5970	5550	5260
req09	17950	10120	10250	14650	5060	10400	8500
	18060	11990	13750	10120	9890	5130	9960
req10	14050	5740	5770	5770	11840	10280	13650
	13910	9800	11110	5790	11210	8640	8730
req11	48550	28290	28920	29210	21320	22770	23790
	48500	29560	28370	28680	21460	22370	22700
req12	63790	34740	34600	34130	26210	26820	22640
	63760	34360	35010	34540	27240	27190	27210
req13	17710	11860	10880	10700	12030	11910	11430
	17770	12480	10080	9290	11250	10860	10590
req14	44880	25670	27100	25180	21080	17480	18580
	45010	25730	25200	26550	20020	21880	19890

Table 7.2: Elapsed times (ms) on an ICL DRS 6000

The framework for cooperation among high-level agents that has been developed for the MaTourA paradigm is a general purpose one which, although it doesn't profit from parallel CLP, may be considered as orthogonal to the latter. It has to be noted, though, that the lower level structuring technique of subagents is clearly based on parallel logic programming execution.

As far as possible extensions of the present technology are concerned, perhaps, a more transparent mechanism for interaction among agents, without explicit message passing, might be more friendly to the application developer. However, this is an issue that needs careful study of what is required and how it may be combined with a parallel CLP environment.

Summarizing the history of the development of MaTourA, it has to be said that most of the effort has been put on the TGA, mainly on the modelling of the constraint satisfaction

	Sun machine		ICL machine			
	S_2	D_2	S_2	D_2	S_3	D_3
req01	1.749	1.102	1.923	1.055	1.964	1.081
req02	1.184	1.126	1.158	1.092	1.200	1.016
req03	1.529	1.110	1.550	1.005	2.243	1.005
req04	1.485	1.106	1.457	1.005	1.958	1.051
req05	1.596	1.065	1.471	1.006	2.088	1.056
req06	1.497	1.102	1.722	1.147	1.979	1.017
req07	1.851	1.046	1.848	1.026	2.611	1.011
req08	1.618	1.075	1.695	1.010	2.023	1.054
req09	2.061	1.326	1.542	1.163	2.307	1.361
req10	1.762	1.091	1.989	1.326	1.321	1.177
req11	1.679	1.052	1.683	1.016	2.168	1.038
req12	1.716	1.068	1.845	1.008	2.438	1.068
req13	1.858	1.125	1.638	1.103	1.565	1.047
req14	1.794	1.076	1.736	1.027	2.274	1.078

Table 7.3: Speedups/Deviations

TCP. It is estimated that almost 40% of the total effort has been given to this issue. Less time, say 20% of the total, has been devoted to other issues, such as problem specification, parallelization, performance debugging and optimization. The remaining 40% of the time has been used for the design and development of the other agents and of the communication framework for the support of the multi-agent systems.

Finally, since sometimes portability is a desirable feature of a system, it should be mentioned that porting MaTourA from one hardware platform to another (e.g. from Sun Sparc machines under SunOS or Solaris to the ICL DRS 6000) has required absolutely no effort. Moreover, during the span of the APPLAUSE project, it has been required to port the application from ElipSys to ECL^{i}PS^e. This has been achieved with surprisingly minimum effort.

Bibliography

[AH85] G. Agha and C. Hewitt. Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism. AI Memo 865, Massachusetts Institute of Technology, 1985.

[Bé94] D. Béja. Recherche d'Optimisation d'une Formation, Etude de Faisabilité. Technical report, Dassault Aviation, Université Paris 6, 1994.

[BBDR+90] U. Baron, S. Bescos, S. A. Delgado-Rannauro, P. Heuzé, M. Dorochevsky, M. Ibáñez-Espiga, K. Schuerman, M. Ratcliffe, A. Véron, and J. Xu. The ElipSys Logic Programming Language. Technical Report DPS-81, ECRC, December 1990.

[BC94] N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 20(12):97–123, 1994.

[BCF95] J. Bellone, A. Chamard, and A. Fischler. Constraint Based Decision Support Systems for Planning and Scheduling Aircraft Manufacturing at Dassault Aviation. In *Proceedings of the International Conference on Improving Manufacturing Performance in a Distributed Enterprise: Advanced Systems and Tools*, pages 109–118, 1995.

[BCP92] J. Bellone, A. Chamard, and C. Pradelles. PLANE, An Evolutive Planning System written in CHIP. In *Proceedings of the International Conference on the Practical Application of Prolog (PAP'92)*, 1992.

[BG88] A. Bond and L. Gasser, editors. *Readings in Distributed Artificial Intelligence.* Morgan Kaufmann Publishers Inc., San Mateo, California, 1988.

[BKW+77] F.C. Bernstein, T. Koetzle, G.J.B. William, E. Jr. Meyer, M.D. Brice, J.R. Rodgers, O. Kennard, T. Shimanouchi, and M. Tasumi. The Protein Data Bank: a Computer-Based Archival File for Macromolecular Structures. *Journal of Molecular Biology*, 112:535–542, 1977.

[BS87] W. Buettner and H. Simonis. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation*, 4:191–205, October 1987.

[BS93] J. Bellone and V. Sarrancanie. Complexity analysis of psap. Dassault applause background document, Dassault Aviation, June 1993.

[BSST87] T.L. Blundell, B.L. Sibanda, M.J.E. Sternberg, and J.M. Thornton. Knowledge-Based Prediction of Protein Structures and the Design of Novel Molecules. *Nature*, 326:347–352, 1987.

[CF94] A. Chamard and A. Fischler. MADE, A Workshop Scheduler System written in CHIP. In *Proceedings of the Second International Conference on the Practical Application of Prolog (PAP'94)*, pages 123–136, 1994.

[CF95] A. Chamard and A. Fischler. Applying CLP to a Complex Scheduling Problem — The MADE System. ESPRIT Project 5291, CHIC, Deliverable D6.5.3. Technical report, Dassault Aviation, 1995.

[CFGG95a] A. Chamard, A. Fischler, A. Guillaud, and D. Guinaudeau. CHIC Lessons on CLP Methodology. ESPRIT Project 5291, CHIC, Deliverable D2.1.2.3. Technical report, Dassault Aviation, Bull, 1995.

[CFGG95b] A. Chamard, A. Fischler, B. Guinaudeau, and A. Guillaud. CHIC Lessons on CLP Methodology. Public CHIC Deliverable Month 48 D.2.1.2.3, Dassault & BULL, February 1995.

[CG90] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course*. MIT Press, Cambridge, 1990.

[CHI94] *CHIP 4.1 Reference Manual*. Cosytec SA, Parc Club Orsay Université, F-91893 Orsay Cedex, France, 1994.

[Col87] A. Colmerauer. Opening the PROLOG III Universe. *Byte Magazine*, 1987.

[CRD94] D.A. Clark, C.J. Rawlings, and S. Doursenot. Genetic Map Construction with Constraints. In R. Altman, D. Brutlag, P. Karp, Lathrop., and D. Searls, editors, *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, pages 78–86. AAAI/MIT Press, 1994.

[CRS+93a] D.A. Clark, C.J. Rawlings, J. Shirazi, L-L. Li, K. Schuerman, M. Reeve, and A. Véron. Solving Large Combinatorial Problems in Molecular Biology Using the ElipSys Parallel Constraint Logic Programming System. *Computer Journal*, 36(4):690–701, 1993.

[CRS+93b] D.A. Clark, C.J. Rawlings, J. Shirazi, A. Véron, and M. Reeve. Protein Topology Prediction through Parallel Constraint Logic Programming. In L. Hunter, D. Searls, and J. Shavlik, editors, *Proceedings of the First International Conference on Intelligent Systems for Molecular Biology*, pages 83–91. AAAI Press, 1993.

[DBH93] B. De Backer and Beringer H. A CLP Language Handling Disjunctions of Linear Constraints. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 550–563, 1993.

[Din86] M. Dincbas. Constraints, Logic Programming and Deductive Databases. In *Proceedings of France-Japan Artificial Intelligence and Computer Science Symposium*, pages 1–27, Tokyo, Japan, October 1986. ICOT.

[DS83] R. Davis and R. Smith. Negotiation as a Metaphor for Distributed Problem Solving. *Artificial Intelligence*, 20(1):63–109, 1983.

[DSR91] Clark. D.A., J. Shirazi, and C.J. Rawlings. Protein Topology Prediction through Constraint Based Search and the Evaluation of Topological Folding Rules. *Protein Engineering*, 4:751–761, 1991.

[DSV87] M. Dincbas, H. Simonis, and P. Van Hentenryck. Extending Equation Solving and Constraint Handling in Logic Programming. In *Proceedings of Colloquium on the Resolution of Equations in Algebraic Structures (CREAS)*, Austin, Texas, USA, May 1987. MCC.

[DVS+88a] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. Applications of CHIP to Industrial and Engineering Problems. In *First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Tullahoma, Tennessee, USA, June 1988.

[DVS+88b] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings on the International Conference on Fifth Generation Computer Systems FGCS-88*, Tokyo, Japan, December 1988.

[ECL94] *ECL^{i}PS^e Extensions User Manual.* ECRC GmbH, Arabellastr. 17, D-81925 München, Germany, July 1994.

[ECL95] *ECL^{i}PS^e 3.5: User Manual*, February 1995.

[Eli93] *ElipSys User Manual for Release Version 0.7*, December 1993.

[EM88] R. Engelmore and T. Morgan, editors. *Blackboard Systems.* Addisson-Wesley Publishing Company, Wokingham, England, 1988.

[Ert94] W. Ertel. On the Defintion of Speedup. In *Proceedings of the 6th International PARLE Conference (Parallel Architectures and Languages Europe)*, number 817 in LNCS, pages 180–191. Springer-Verlag, 1994.

[FHK+92] T. Frühwirth, A. Herold, V. Küchenhoff, L. Le Provost, P. Lim, E. Monfroy, and M.G. Wallace. Constraint Logic Programming - An Informal Introduction. In G. Comyn, N.E. Fuchs, and M. Ratcliffe, editors, *Logic Programming in Action, Second International Logic Programming Summer School, LPSS'92*, volume 636 of *Lecture Notes in Artificial Intelligence*, Zürich, Switzerland, September 1992. Springer Verlag.

[Fru95] T. Fruehwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraints: Basics and Trends.* Springer LNCS, 1995. to appear 1995.

[Gal85] H. Gallaire. Logic Programming: Further Developments. In *Proceedings of the IEEE Symposium on Logic Programming*, Boston, USA, July 1985.

[Ger94] C. Gervet. Conjunto: Constraint Logic Programming with Finite Set Domains. In M. Bruynoghe, editor, *Proceedings of ILPS'94.* International Logic Programming Symposium, 1994.

[Hew88] C. Hewitt. Offices Are Open Systems. *ACM Transactions on Office Information Systems*, 4(3):271–287, April 1988.

[HKS+94] C. Halatsis, I. Karali, P. Stamatopoulos, C. Mourlas, D. Gouscos, and C. Fouskakis. ElipSys Assessment. APPLAUSE Project Deliverable D.WP3.6, University of Athens, June 1994.

[Hol95] C. Holzbaur. OFAI clpq(Q, R) Manual, Edition 1.3.3. Technical report, Austrian Research Institute for Artificial Intelligence, 1995.

[Hon92] H. Hong. Non-linear Constraint Solving over Real Numbers in Constraint Logic Programming (Introducing RISC-CLP). Technical Report D.1.1, RISC, Linz, February 1992.

[HR85] B. Hayes-Roth. A Blackboard Architecture for Control. *Artificial Intelligence*, 26:251–321, 1985.

[HSK+94] C. Halatsis, P. Stamatopoulos, I. Karali, C. Mourlas, D. Gouscos, D. Margaritis, C. Fouskakis, A. Kolokouris, P. Xinos, M. Reeve, A. Véron, K. Schuerman, and L.-L. Li. MaTourA: Multi-agent Tourist Advisor. In *Proceedings of the International Conference on Information and Communication Technologies in Tourism*, pages 140–147. Springer-Verlag, 1994.

[HSKG95] C. Halatsis, P. Stamatopoulos, I. Karali, and D. Gouscos. ElipSys (ECL^{i}PS^e) Support of Multi-agent Systems. APPLAUSE Project Deliverable D.WP3.8B, University of Athens, August 1995.

[HSKM92] C. Halatsis, P. Stamatopoulos, I. Karali, and C. Mourlas. PETINA — PErsonalized Tourist INformation Advisor: Final Report. EDS Project Deliverable EDS.WP.9E.A009, University of Athens, July 1992.

[HSM+93] C. Halatsis, P. Stamatopoulos, D. Margaritis, I. Karali, C. Mourlas, D. Gouscos, and C. Fouskakis. Tool Assessment. APPLAUSE Project Deliverable D.WP3.4, University of Athens, May 1993.

[HSP+93] C. Halatsis, P. Stamatopoulos, Z. Palaskas, I. Karali, C. Mourlas, D. Gouscos, D. Margaritis, and C. Fouskakis. MaTourA Specification. APPLAUSE Project Deliverable D.WP3.2, University of Athens — Expert Systems International, May 1993.

[Huh87] M. Huhns, editor. *Distributed Artificial Intelligence*. Pitman, London, 1987.

[JL87] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *POPL'87*, 1987.

[JM94] J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, pages 503–581, 1994.

[LRS+93a] L.-L. Li, M. Reeve, K. Schuerman, A. Véron, J. Bellone, C. Pradelles, A. Kolokouris, T. Stamatopoulos, D. Clark, C. Rawlings, J. Shirazi, and G. Sardu. APPLAUSE: Application & Assessment of Parallel Programming Using Logic. In *Proceedings of the 5th International PARLE Conference*, pages 756–759, 1993.

[LRS+93b] L.-L. Li, M. Reeve, K. Schuerman, A. Véron, J. Bellone, C. Pradelles, Z. Palaskas, T. Stamatopoulos, D. Clark, S. Doursenot, C. Rawlings, J. Shirazi, and G. Sardu. APPLAUSE: Applications Using the ElipSys Parallel CLP System. In *Proceedings of the 10th International Conference on Logic Programming*, pages 847–848, 1993.

[LW93] T. Le Provost and M.G. Wallace. Generalised constraint propagation over the CLP Scheme. *Journal of Logic Programming*, 1993.

[MF88] A.J. Murzin and A.V. Finkelstein. General Architecture of the α – Helical Globule. *Journal of Molecular Biology*, 204:749–769, 1988.

[MS92] Micha Meier and Joachim Schimpf. An Architecture for Prolog Extensions. In *Proceedings of the 3rd International Workshop on Extensions of Logic Programming*, Bologna (Italy), 1992.

[MS94] S. Mudambi and J. Schimpf. Parallel CLP on Heterogeneous Networks. In MIT Press, editor, *Proceedings of the International Conference on Logic Programming ICLP'94*, Santa Margherita Ligure, Italy, June 1994.

[Mud94] S. Mudambi. ElipSys Evaluation II. Restricted APPLAUSE Deliverable Month 24 - D.WP4.ECRC.4C1.3, ECRC, June 1994.

[PA92] Z. Palaskas and T. Athanasopoulos. MaTourA User Requirements. APPLAUSE Project Deliverable D.WP3.1, Expert Systems International, November 1992.

[Per94] G. Perrat. Optimisation de cursus de formation à l'aide de la Programmation Logique avec Contraintes. Technical report, Dassault Aviation, Ecole des Mines de Nancy, 1994.

[PM94] S. Prestwitch and S. Mudambi. Cost-parallel Branch and Bound in CLP. In *Proceedings of ILPS 1994 Post-conference Workshop on Constraint Languages and Systems*, pages 141–149, 1994.

[PM95] S. D. Prestwich and S. Mudambi. Improved Branch and Bound in Constraint Logic Programming. In U. Montanari, editor, *Proceedings of Constraint Programming 95*, LNCS. Springer Verlag, September 1995. (appeared also as ECRC Technical Report ECRC-95-19).

[Pre92] S. Prestwich. ElipSys Programming Tutorial. Public APPLAUSE Deliverable Month 6 - D.WP4.ECRC.4.4a, ECRC, November 1992.

[Pre93a] S. Prestwich. ElipSys Programming Tutorial. Technical Report ECRC-93-2, ECRC, January 1993.

[Pre93b] S. Prestwich. Parallel Speedup Anomalies and Program Development. Technical Report ECRC-93-12, ECRC, September 1993.

[Pre94a] S. Prestwich. A Note on Calculating Parallel Speedup. Technical report, ECRC GmbH, Arabellastr. 17, D-81925 München, Germany, 1994.

[Pre94b] S. Prestwich. A Tutorial on Parallelism and Constraints in ECL^{i}PS^e. Technical report, ECRC, 1994.

[Pre94c] S. Prestwich. On Parallelisation Strategies for Logic Programs. In *Proceedings of the International Conference on Parallel Processing '94*, number 854 in LNCS, pages 289–300. Springer-Verlag, 1994.

[SHL+92] C. Skelton, C. Hammer, M. Lopez, M. Reeve, P. Townsend, and K.-F. Wong. EDS: A Parallel Computer System for Advanced Information Processing. In *Proceedings of the 4th International PARLE Conference*, pages 3–18, 1992.

[SKH92] P. Stamatopoulos, I. Karali, and C. Halatsis. PETINA — Tour Generation Using the ElipSys Inference System. In *Proceedings of the 1992 ACM Symposium on Applied Computing*, volume 1, pages 320–327, 1992.

[SKH93] P. Stamatopoulos, I. Karali, and C. Halatsis. A Tour Advisory System Using a Logic Programming Approach. *Applied Computing Review*, 1(1):18–25, 1993.

[SMH94] P. Stamatopoulos, D. Margaritis, and C. Halatsis. Extending a Parallel CLP Language to Support the Development of Multi-agent Systems. In *Proceedings of the 1994 ACM Symposium on Applied Computing*, pages 410–414, 1994.

[Smi80] R. Smith. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, December 1980.

[Ste90] W. R. Stevens. *UNIX Network Programming.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Sys88] Metier Managment Systems. Introduction dans la gestion de project. Technical report, Metier Managment Systems, November 1988. ARTEMIS is now a product of Lucas Management Systems.

[TG89] W.R. Taylor and N.M. Green. The Predicted Secondary Structure of the Nucleotide Binding Sites of six Cation Transporting ATPases Leads to a Probable Tertiary Fold. *Journal of Biochemistry*, 179:241–248, 1989.

[V93] A. Véron. Disjunctive Scheduling in Elipsys. Technical report, ECRC GmbH, Arabellastr. 17, D-81925 München, Germany, 1993.

[Van89a] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series. MIT Press, Cambridge, Ma, 1989.

[Van89b] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming: Preliminary Results of CHIP within PEPSys. In *Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989.

[vH89] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, Cambridge, Massachusetts, 1989.

[VSRL93] A. Véron, K. Schuerman, M. Reeve, and L-L. Li. Why and How in the ElipSys OR-parallel CLP system. In Reeve Bode and Wolfe, editors, *PARLE '93, Parallel Architectures in Europe*, pages 756–759. Springer-Verlag, 1993.

[XSG+94] P. Xinos, P. Stamatopoulos, D. Gouscos, I. Karali, and J. Paine. MaTourA Description Manual. APPLAUSE Project Deliverable D.WP3.7B, Expert Systems International, December 1994.