TR-594

Parallel Forward Checking
First part

by
Bernard Burg

September, 1990

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

Parallel Forward Checking

First part

ICOT
1-4-28 Mita, Minato-ku, Tokyo 108, Japan

Bernard Burg

July, 1990

Abstract

One of the most popular ways to conceive parallel algorithms, is to tailor a version having high inherited parallelism on one processor, and to speed it up on several processors. Thus the published speedup are often impressive, but the raw speed performances turn out to be low, and highly efficient sequential algorithms are sometimes outperforming the parallel algorithms.

In aiming to avoid such disappointments, we started this study by looking for the most efficient sequential algorithm to perform *Constraint Satisfaction Problems* in finite space. The algorithms responding to this criteria are *Forward Checking*. We describe their principle, and focus of the way we transformed the sequential algorithms into their parallel versions. This report describes the four *Parallel Forward Checking Problem Solvers* we implemented, two or them make one-solution search by using *speculative AND-parallelism*, the two others do exhaustive search in the frame of *OR-parallelism*. A user guide explains all their parameters and the way to conceive an application using them. The important mechanisms are taken apart, the limitations and error messages are explained.

As the main purpose of this study is to provide efficiency on a parallel computer, results are demonstrated on well known problems: the queens, the zebra, scheduling and mazes. They have been run on the Multi-PSI[8], full results are reported: speedup, raw speed and reductions. The load balancing between the processors is studied to explain the speedups. Super-linear speedup have been reached as well in *speculative AND-parallelism* as is *OR-parallelism*.

Contents

List of Figures

List of Tables

Introduction

Constraint Satisfaction Problems CSP is an accurate tool to perform efficient search in large search spaces. Many papers describe the theory of CSP's[12] and outline efficient methods to solve them[11]. Implementations lead sometimes to bitter disappointments[13], especially in the parallel environment. Thus we chose a simple but reliable problem solving method, reported by serious papers[5, 4]: namely *Forward Checking*. We implemented at first efficient sequential versions of these *Problem Solver*, then we extended the study to the parallel environment. The parallelism of our algorithms is expressed by constraints thus sequential and parallel implementations keep the same mechanisms.

The first part of this report has been divided into three chapters: the first of them explaining the principle of the *Parallel Forward Checking*, it shows the implementation techniques we used to achieve efficiency and gives a user guide explaining how to realize an application using the *Parallel Forward Checking*.
The second chapter focus on applications and performances. Several applications are shown: the queen problem, the zebra problem, scheduling and mazes are solved with exhaustive search as well as with unique solution search. All performance measures were obtained on the Multi-PSI.
The third chapter studies point by point the results and draws conclusions from this work, giving some advises to future users of this programs.

The second part of this report will describe a new load balance technique for exhaustive solution search, give the related results, and open the source code of both, *Problem Solver* and applications.

Acknowledgment

I have to thank several people for their help, or the discussions we had, concerning this work. First of all, I would like to thank Takashi Chikayama for his infinite kindness, his constant support and the fruitful discussion we had. He is the author of the efficient sequential queen algorithm, Q_SFCH algorithm reported in the related work. Katsuto Nakajima was an excellent *talking optimizer* for fine tuning of the programs, David Hawley was a talented critics; both of them showed interest to the work and made very useful suggestions on the early draft of this report. Last, but not least, Satoshi Terasaki translated these *Parallel Forward Checking Problem Solvers* from KL1 into ESP. His ESP versions are outperforming the KL1 version running on a single processor.

Chapt

Parallel Forward Checking Problem Solvers

The principle of forward checking is one of the most efficient procedure for solving CSP's, section 1.1 outlines its principles. They are extended to the parallel environment in section 1.2, in an original manner so to obtain *Parallel Forward Checking*. Our methodology of parallelism gives a warranty of efficiency, the parallel algorithms run at least as fast as the optimal sequential algorithm. As these notions are quite abstract so far, we illustrate our discourse with the archetypal example of the queen problem in section 1.3. In section 1.4, some insights of the implementation techniques are drawn so to favor a deep understanding of those problem solvers, showing their limitation, force and weakness[1]. Finally, we give in section 1.5 the user guide of the Parallel Forward Checking Problem Solvers.

1.1 Principle of forward checking

1.1.1 Definitions

We introduce the definition of a CSP problem, and the definition of the two kinds of constraints used.

Constraint Satisfaction Problem CSP

A constraint satisfaction problem CSP can be defined as follows: given a set of variables $x_1, \ldots, x_n$ and associated with each value of variable x_i a domain D_i of values[2]. Furthermore, on some subsets of the variables constraints are given, limiting possible value tuples for those variables. A solution of the CSP is a n-tuple of values $(a_1, \ldots, a_n) \in D_T = D_1 \times \ldots \times D_n$, which simultaneously satisfies all given constraints.

Static constraint

The *static constraints* $\mathcal{SC}$ are used during the initialization of the domains of each variable. These constraints reduce the domain of value of the variables, before the start of the search algorithm. They may be formalized as follows:

$$\mathcal{SC}(x_i, D_1 \times \ldots \times D_i \times \ldots \times D_n) = D_1 \times \ldots \times D_{i-1} \times D'_i \times D_{i+1} \times \ldots \times D_n$$
$$\text{with } x_i \in D_i \text{ and } |D_i| \geq |D'_i|$$

As we will see later, these *static constraints* are used in two ways, to define some problems (see section 1.5) and to introduce parallelism (see section 1.2).

Dynamic constraint

The *dynamic constraints* $\mathcal{DC}$ used in CSP are functions linking an instance of a variable x_i to other variables, and reducing their domain of values. The *dynamic constraints* are used by the core of the CSP to propagate constraints linking two variables. This is:

$$\mathcal{DC}(x_i, x_j, D_1 \times \ldots \times D_i \times \ldots \times D_n) = D_1 \times \ldots \times D_{j-1} \times D'_j \times D_{j+1} \times \ldots \times D_n$$
$$\text{with } x_i \neq x_j,\ x_i \in D_i,\ x_j \in D_j \text{ and } |D_j| \geq |D'_j|$$

Thus, propagating a constraint reduces the domain of the non-determinacy D_T, since D'_j is smaller than D_j.

[1] The section 1.4 may be omitted in a first lecture of this report.

[2] The indexes of the variables are in there range of $1 \ldots n$ in the sections 1.1 1.2 1.3, to ease the notations, but in sections 1.4 1.5 the variables are encoded in $0 \ldots (n-1)$, as in the programs

1.1.2 Simple forward checking SFCH

The algorithm starts by defining the domains of non-determinacy D_T in line 1. The initialization applies the static constraints $\mathcal{SC}$ to these initial domains of value. Then, simple forward checking is to take in turn each of the variables and to propagate their dynamic constraints $\mathcal{DC}$ to all the non-determined variables. If the domain of a variable in D_T is empty, then no solution is available, as shown in line 9; both i and j loops are left, and SFCH backtracks to find another solution. Otherwise, the checked instance x_i is added to the determined set S and the domain of the checked variable D_i is removed from the domain of non-determinacy D_T, respectively in lines 10 11.
This loop is repeated for all variables. Finally, a solution of the CSP problem is stored in the S set of determined values. The skeleton of simple forward checking is shown in Table 1.1. This principle may be used is the frame of

```
1    D_T = D_1 × ... × D_n
2    for i = 1...n
3        D_T = SC(x_i , D_T)
4    S = ∅
5    for i = 1...n
6        x_i ∈ D_i
7        for j = i...n
8            D_T = DC(x_i , x_j , D_T)
9            when |D_T| = 0, exit(S = ∅)
10       S = S ∪ {x_i}
11       D_T = D_{i+1} × ... × D_n
```

Table 1.1: Outline of SFCH

depth first search in aiming to find a solution of the CSP, or to find all the solutions. To do so, the line 6 has to be more precise and to give exactly the choice of the instance of the x_i element. Some material added to allow backtracks and the merge of the results have not been described for sakes of simplicity.

The main advantage of SFCH is its simplicity, it consists in a loop, checking in turn the variables from 1 to n, regardless to their domain of values given by the constraints.

1.1.3 Generalized forward checking GFCH

Instead of using the constraints in a fixed order, it seems interesting to reorder them dynamically, to check at first the most constrained variable x_i. Thus, if there is no solution for the CSP, the algorithm will detect this impossibility as soon as possible, avoiding several backtracks.

In comparison to SFCH, GFCH adds some more work to perform; after the constraint propagation of each variable, GFCH has to sort the non-deterministic variables in D_T according to the size of their domains (see line 5). The one having the smallest domain will be checked first. The structure shown in Table 1.2 is similar to the one of SFCH

```
1    D_T = D_1 × ... × D_n
2    for i = 1...n
3        D_T = SC(x_i , D_T)
4    S = ∅
5    for D_i = min_{α∈D_T} |D_α|
6        x_i ∈ D_i
7        for α ≠ i
8            D_T = DC(x_i , x_α , D_T)
9            when |D_T| = 0, exit(S = ∅)
10       S = S ∪ {x_i}
11       D_T = D_T / D_i
```

Table 1.2: Outline of GFCH

Theoretically, GFCH should be more efficient than SFCH because it leads more efficiently to the solutions. However, the implementations face an additional work to perform in line 5, namely defining the size of the domains and sorting them. It is hard to predict which of SFCH of GFCH will be the most efficient, since it depends on the nature of the constraints we deal with, the size of the search spaces, the branching factor of the problem etc.

1.1.4 Unique solution search SFCH1 GFCH1 versus exhaustive search SFCH GFCH

Until now, we did not differentiate unique solution search algorithms from exhaustive solution search ones, since the outlines presented in Tables 1.1 and 1.2 are valid for both because they represent only the innermost search loop looking for a single solution.

We implemented four algorithms:
The unique solution search algorithms SFCH1 and GFCH1, repeat this loop until a solution is found, conversely the exhaustive search algorithms SFCH and GFCH explore the whole search space and collect all the results.

From now on, the meaning of SFCH and GFCH means exhaustive search according to respectively Simple and Generalized Forward Checking, SFCH1 and GFCH1 are the names of the unique solution search algorithms using respectively the same principles.

1.2 Forward checking in parallel

The art of parallel programming consists in finding a good load balance between processors without dramatic increase of inter-processor communications. This remains the key of the success, especially on the Multi-PSI.

Forward checking, as we described it earlier, is basically a sequential method; one variable is checked after the other, even worse, in GFCH and GFCH1 there is an ideal order to do so. Nevertheless, our definition of CSPs makes reference to finite domains, thus parallelism is found in another dimension that the principle of the algorithm, namely the search space. To put forward checking in parallel, is simply to split the search space into subspaces, one for each of the processors.

This operation is quite natural and easy to express in form of *static constraints* $\mathcal{SC}$. It is to add new constraints to the initial domains of the variables, so that each of the processors will perform the search in a different subspace, the union of those subspaces being the complete search space. Back to the Tables 1.1 and 1.2, *Parallel Forward Checking* is to give the following $\mathcal{SC}$ relation:

$$\mathcal{SC}(x_i\ ,\ D_1 \times \ldots \times D_n) = D_{1_{pro}} \times \ldots \times D_{n_{pro}}$$
$$\text{with} \sum_{pro} \left(\prod_{i=1}^{n} D_{i_{pro}} \right) = D_1 \times \ldots \times D_n$$

Let us emphasize the simplicity of this static load balance method, requiring no overwork nor communication between processors. The efficiency of this load balancing relies on the ability to split the search space into homogeneous parts. Even when running on one processor, our *Parallel Forward Checking* remains a very efficient algorithm, in contrast to many other parallel algorithms, where pruned search in parallel implies sometimes overwork to be done[1].
Last but not least, the parallel algorithm is totally compatible with the sequential one, as a consequence every theoretic deduction holds in the parallel environment; also in a more down-to-earth mind, the additional software written to transform the sequential algorithms into parallel ones is of very small size thanks to the use of FLIB's[2] facilities.

Nevertheless, this parallel algorithm has restrictions. When used in a highly constraint domain, there may be only one solution to the CSP problem. In such a case there is a low, or no parallelism in the problem, since the forward checking leads very efficiently to the solution. The parallel implementation cannot do it in a better way, whatever its principle.
Another drawback is that static load balancing cannot give any warranty for a smart load balancing. Even with equal sized search spaces of each processor, the work to perform may be different since some search spaces may exhibit a higher solution density than others. Splitting the search spaces remains a challenge.

1.2.1 OR-Parallelism for exhaustive search, SFCH and GFCH

Each of the processors searches solutions of the CSP in a subspace of the search space, the solution of the problem is the union of the solutions found by all the processors, thus exhaustive solution search is done in the frame of

OR-Parallelism.

1.2.2 Speculative AND-Parallelism for unique solution search, SFCH1 and GFCH1

Here too, each of the processors searches solutions of the CSP in a subspace of the search space. As soon as one of the processor finds a solution of the problem, it sends a message to the other fellows, to stop their searches. The principle of parallelism we adopted for unique solution search is the *speculative AND-Parallelism*.

1.3 Illustrative example: the queen problem

After these abstract sections explaining the principles of the *Parallel Forward Checking*, it seems good to follow an example and illustrate our talk. The problem of the queens, which consists to place n queens on an $n \times n$ board, so that no one is attacked by another; is well suited to illustrate the *Parallel Forward Checking*. The constraints and the parallel features we introduced are easy to represent in the queen problem, since the variables of the queen problems are the columns, and their domain of values are the lines. As a consequence, the board representation is the most explicit and natural.

1.3.1 Simple forward checking SFCH, and SFCH1

To make things more explicit, we follow the 5 queens example, the reader will generalize with no difficulty. There are 5 variables, one for each column. At the begin of the algorithm, the domain of non-determinacy of each variable is set to all the possible line values, $D_T = [1, 2, 3, 4, 5]^5$, and $S = \emptyset$. Then, in line 6 of Table 1.1, an instance of x_1 is chosen, say 1. Immediately, all the inconsistent values of $x_2 \ldots x_5$ are removed from their possible set, by the $\mathcal{DC}$ constraints. Indeed, x, and the diagonals are removed from the D_T domain, and $x_1 = 1$ is added to S. Therefore, at this time, the situation is shown in Figure 1.1. In the next step, the values 1 and 2 will not be considered for x_2 since they are not

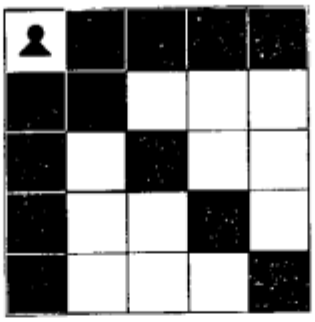

Figure 1.1: Five-queens problem after 1 choice

included in D_T:

$$D_T = D_2 \times D_3 \times D_4 \times D_5$$
$$D_T = [3, 4, 5] \times [2, 4, 5] \times [2, 3, 5] \times [2, 3, 4]$$

The next step chooses for example, the value 3 for x_2, thus the non-deterministic domain D_T becomes, in Figure 1.2:

$$D_T = D_3 \times D_4 \times D_5$$
$$D_T = [5] \times [2] \times [2, 4]$$

Now x_3 and x_4 have domains reduced to singletons. By checking and propagating their constraints, the x_5 domain is as well reduced to a singleton. The problem of the five queens, is solved with 2 choices and without backtrack.

1.3.2 Generalized forward checking GFCH GFCH1

To explain the generalized forward checking, we have to take a bigger size example, the eight queens. Consider now the problem after 3 choices, in Figure 1.3. The domain D_6 is reduced to a singleton, thus generalized forward checking will test it at first, because it is the most constraint variable. This allows to propagate the constraints without making

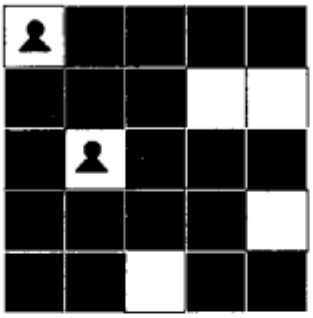

Figure 1.2: Five-queens problem after 2 choices

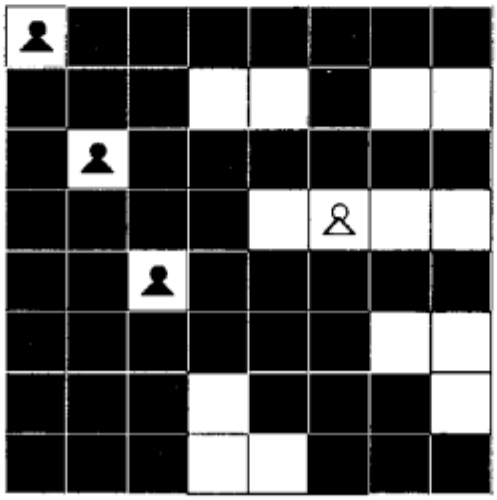

Figure 1.3: Eight-queens problem after 3 choices

a choice. It reduces significantly the number of backtracks to be performed. Nevertheless, one has to notice that forward checking has to update the domain of each non-determined variable to find the most constraint of them.

1.3.3 Forward checking in parallel

As stated earlier, two kinds of parallelism are used, OR-Parallelism to perform exhaustive search in SFCH and GFCH; and speculative AND-Parallelism to make unique solution search in SFCH1 and GFCH1. Nevertheless, the load balancing used in both searches is the same, we explain it below.

Parallel implementations keep the principle of the sequential ones. The initialization only changes, by addition of *static constraints* $\mathcal{SC}$ (see 1.2). As the Multi-PSI has a number of processors in power of two, we made a simple load balancing, working only for the powers of two processors. The idea is quite elementary. One domain of values is split into two equal parts. For example:

$$D_i = [1, 2, 3, 4, 5, 6, 7, 8 \ldots n]$$
is split into:
$$D_i^{odd} = [1, 3, 5, 7 \ldots (n-1)] \quad D_i^{even} = [2, 4, 6, 8 \ldots n]$$

Processing the queens with 2 processors is to split one of the column (which is represented by the domain of a variable) in two equal parts. So making forward checking on 2 processors, is to run exactly the same algorithm on each processor, each of them starting with some additional constraints, see Figure 1.4.

In the same manner, processing the queens with 4 processors is to split two columns rather then one. The same

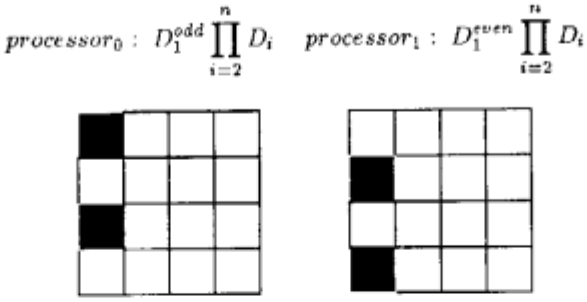

Figure 1.4: Load balance for 2 processors

example is done by 4 processors in Figure 1.5.

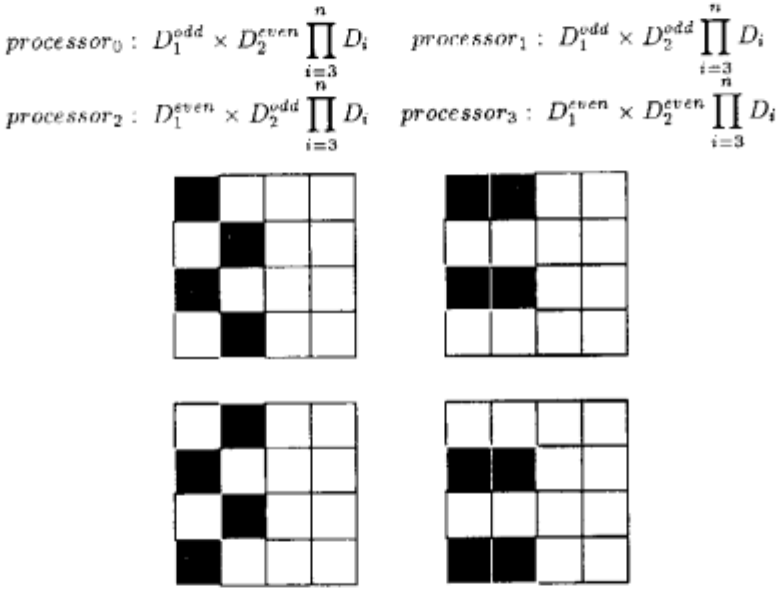

Figure 1.5: Load balance for 4 processors

And so on, to split the problem on more processors, more columns are used. This simple load balancing is only working for the powers of 2, but this is exactly what is required by the Multi-PSI.

1.4 Implementation

1.4.1 Architecture of the modules

The different programs we wrote, are each in a module, having the name of the program, namely **`sfch`**, **`sfch1`**, **`gfch`**, **`gfch1`**. All these modules access to some common utilities in the module **`fch`**. These softwares have been built on top of the FLIB[2] library. They run as well on PDSS, PSI and Multi-PSI.

The *Problem Solvers*, SFCH, SFCH1, GFCH, GFCH1, access to the module containing the application, and evaluate the predicate **`constraints/2`**. The name of this module is the second parameter given to the problem solver.

1.4.2 Techniques to reach efficiency

By its principle, Forward Checking is to apply constraints to the non-determinacy domains D_T. In other words, this is to make the intersection between the non-determinacy domain D_T and the domain allowed by the constraint. Thus, to be efficient, we have to find an encoding allowing fast intersections between domains.

One of the most popular technique to do so, is to encode the two domains into words, and to make the logical and between these words. We adopted this bit-encoding, which turns out to be several degree of magnitude faster than the naive encoding, however, we introduce by this way a limitation, since the bit-words are 32 bits long, this program is limited to treat domains of value restricted to 32 elements.

Writing and debugging programs dealing with bit-encoding is not particularly delightful. To avoid this pain to the user, we defined the format of the application dependent constraints as a list of constraints (see section 1.5). As a consequence the user may write the application dependent predicate **constraints** in a naive way, generating the lists of constraints. These constraints are then compiled into efficient bit encoding by the problem solver. This operation is done once, during the initialization, hence it affects little the performances of the problem solver, provided a large search space.

1.4.3 SFCH and SFCH1: basic implementation

These program implementations transcribe with fidelity the outline shown in Table 1.1. The domains of the variables are encoded into bit-words, thus it is easy to perform the intersections of domains, as well as to check if the domains are empty. The solution of a hash-coding table has been adopted, so to check is the domain of non-determinacy D_T contains a unique element. In this case there is no more non-determinacy, on this variable, and the algorithm continues its work with the remaining elements of D_T.

1.4.4 GFCH and GFCH1: basic implementation

These program transcribe as well with fidelity the outline shown in Table 1.2. The domains of the variables are encoded into bit-words, thus it is easy to perform the intersections of domains, as well as to check if the domains are empty. But the main difference with the *Simple Forward Checking*, is that the *Generalized Forward Checking* has to compute the size of the variables domains, so to find the smallest of them, which determines the order to check the variables. The bit-encoding of the domains makes this operation difficult, so after each application of a constraint to a domain of a variable, we have to count the number of possibilities contained in this new domain, this is to count the number of bits of a word. This operation, called *Domain → Size conversion* has to be highly efficient, since it appears in the innermost loop of the algorithm.

Domain → Size conversion, **naive approach**

The naive approach to count the bits of a word, is to proceed by bit-shift operation, counting all the bites, the one after the other. As we see in Table 1.3, the complexity of this operation

```
naive_bit_sum(Domain, DomSZ, Res):-
     DomSZ > 1,
     NDomain := Domain >> 1,
     RLoc := 1 /\ Domain|
          Res:= RLoc + Res1,
          naive_bit_sum(NDomain, ~(DomSZ-1), Res1).
naive_bit_sum(Domain, 1, Res):- true|
          Res=Domain.
```

Table 1.3: Naive *Domain → Size conversion*

is linear in the length of the words, $DomSZ$. It seems very inefficient to use this method in one of the innermost loops of the algorithm, and maybe the whole efficiency gained by bit-encoding will be doomed by this *Domain → Size conversion*.

Domain → Size conversion, **efficient approach**

The most efficient way to make this conversion, is to store it into a table, thus conversion is reduced to the time of access to this table. The main problem of this solution is the size of this conversion table being in 2^{DomSZ}. As $DomSZ$ is up to 32, we have to imagine another solution since we cannot store a table of 2^{32} elements.

Domain → Size conversion, **implementation**

We implemented a compromise solution between the two previous ones. We store a table containing the conversion of slices of words, in the length of $\frac{DomSZ}{WZ}$ bites. This table contains $2^{\frac{DomSZ}{WZ}}$ elements. Then, the domain words are decoded by checking their WZ slices, like in Table 1.4, where **Table** is the table of the slices conversion, **DGNB** is the number if bites of the slices, and **BMask** is the mask having its DGNB lower bites set to 1.

```
bit_sum(Table, Domain, DomSZ, BMask, DGNB, Res):-
     DomSZ > DGNB,
     NDomain := Domain >> DGNB,
     Index := BMask /\ Domain,
     vector_element(Table, Index, RLoc)|
          Res:= RLoc + Res1,
          bit_sum(Table, NDomain, ~(DomSZ-DGNB), BMask, DGNB, Res1).
bit_sum(Table, Domain, DGNB, _, DGNB, Res):-
     vector_element(Table, Domain, Res1)|Res1=Res.
```

Table 1.4: Implementation *Domain → Size conversion*

As a consequence, the user of our programs may adapt the parameter WZ according to both, the machines memory-size, and the required memory to solve a given problem. Of course, the more memory we use, the faster the conversion of the words.

The first element of a domain

By carrying the same idea as previously, we implemented as well a Table recording the first bit of a word. This operation is used in the GFCH and GFCH1, when the algorithm chooses a solution, which is an element in the domain of the possible ones. The encoding technique takes exactly the same spirit as for the *Domain → Size conversion*. Thus a second table is memorized by the program.

1.4.5 The parallelism implementation

Load balancing

The parallel version of SFCH, SFCH1, GFCH, GFCH1 use the same static load balancing. It is to split the first variables into two parts, as shown in section 1.2. This load balancing method is performed automatically by the program, which generates the required static constraints $\mathcal{SC}$ and applies them over the constraints generated by the application.

Type of parallelism

SFCH and GFCH perform the exhaustive search of the solutions by using OR-parallelism, conversely SFCH1 and GFCH1 search for a unique solution in the frame of speculative AND-parallelism. Both types of parallelism have been implemented by use of FLIBs parallel facilities.

Data spreading

FLIB's list spreading has been chosen to spread the data amongst the processors, since the data transmitted to the processors is of very small size, it consists only of the parameters given to the *Problem Solver*. Then, each processor generates locally the constraints of the problem, compiles them into bit-encoding, adds the *Static Constraints* making the load balancing between processors, and finally runs the Problem Solving algorithm. The results of this algorithm are send to the $processor_0$ of the Multi-PSI. If the parameter **ResCont = number**, the result send by the processors

is the number of local solutions; in the case of **ResCont = all**, the result transmitted by the processors is the list of the solutions found locally.

Comments on the parallel implementation

Some options have been taken during the parallel implementation, let us clarify them:

- Static load balancing is not proved to be accurate, however our parallel implementation running on one processor pays a negligible overhead compared to the optimal sequential algorithm.
- The parallelism has been designed to minimize the communications between processors, which seems to be the main bottleneck on the Multi-PSI according to our experience. Thus the generation and compilation of the constraints are done sequentially, in redundancy on all processors.
- Since the generation of the constraints is an application dependent program written by the end-user, is was important to ease their implementation. Thus we defined **constraints** as a sequential program, maybe written in a naive style.
- It is supposed that the time to perform the constraint generation is negligible before the time to perform the search.

1.4.6 Constraint compilation

The constraint compilation is the program in charge to transform the list of constraints, generated by the application dependent program, into efficient encoding used by the core of all the problem solvers. By the way, the constraints compilation transforms naive encoding into bit encoding.

Internal representation of the constraints

The constraints are represented by a 3 dimensional **constraint-table**, (vectors in KL1). The first dimension represents the variables giving the constraints, the second dimension their instances, the third the variable having new constraints. Thus the dimension of the table is in $VarNb \times DomSZ \times VarNb$.

The structure of this table is similar to the structure of the constraints given in section 1.5. The following constraint: $[Var_i, Inst_i, [Var_j|DomRes_j]]$ meaning that if variable Var_i is instantiated to $Inst_i$, then variable Var_j gets the constraints expressed by $DomRes_j$, will be stored in the **constraint-table** at $(Var_i, Inst_i, Var_j)$.

The table initialization

At the begin, all elements of the **constraint-table** are instantiated to the full domain, represented by:

```
((1 << DomSZ)-1)
```

The constraints

Compiling the constraints, is to take the $DomRes_j$ list expressing the constraints and to transform it into a bit-mask, as shown in Table 1.5.

```
pattern_mask(['+'], _, Res):- true|
          Res := 0.
pattern_mask([Pos|B], Mask, Res):-
     LocMa:= 1 << Pos,
     Mask1:= LocMa \/ Mask|
          pattern_mask(B, Mask1, Res).
pattern_mask(□, Mask, Res):- true|
          Res := Mask xor (-1).
```

Table 1.5: Basic constraint compilation

The new domain for the variable (Var_i, $Inst_i$, Var_j) in `constraint-table`, is obtained by a logical-and between the old domain stored, and the bit-Mask made in Table 1.5.

The program performs the conjunction of the constraints given by application programs.

The distinction we introduced between *Static Constraints* and *Dynamic Constraints* does not exist during the compilation; static constraints are stored in the positions (Var_i, 0, Var_i), since this position cannot be taken by dynamic constraints, according to their definition in section 1.1.

1.5 User guide

This section describes the *Parallel Forward Checking Problem Solvers* we developed, and explains how to use them. As the problem solvers have to handle some specific problems, we show in the next subsection the general framework for such interactions. Then the reader finds more detailed explanation of the programs and their parameters, the manner to express the constraints and eventually the limitations of our current implementation.

1.5.1 General framework for application programs

The *Problem Solvers* have to interact with an application program. The aim of the application program is to generate a list of constraints to be used by the *Problem Solvers*. To do so, the user has to define a module for its application. This module calls one or several *Problem Solvers* and gives them as second parameter, its own module name, see Table 1.6.

```
:- module my_application.
:- public sfch/4, constraints/2.

% to start the problem solver
sfch(VarNb, DomSZ, ProNb, Result):- true|
    sfch:run({VarNb, DomSZ}, 'my_application', 'number', ProNb, Result).

% generation of the constraints
constraints({VarNb, DomSZ}, Constraints):- true|
% static constraints
% dynamic constraints
    Constraints=[].
```

Table 1.6: The application program

In aiming to generate the constraints, the *problem solver* calls the predicate `constraints/2` in the given module. This predicate is the responsibility of the user writing the application program. The first parameter transmitted to the `constraints` predicate, is the vector `{VarNb, DomSZ}`, respectively describing the number of variables and the size of their domain of values. The second argument of the `constraints` predicate is the list of the generated constraints, describing the application. When the constraints are generated, the *Problem Solver* compiles them into a more efficient representation. Then the search algorithm is ready to start its work, as described in section 1.1.

1.5.2 Description of the programs

The following programs share the same parameters, there is just an additional parameter for GFCH and GFCH1, besides this detail, they are totally compatible, using the same constraints, and formatting the output in the same manner. So it is quite simple to switch from one problem solver to another. The parameters are explained in next section.

SFCH

This program makes the exhaustive search of solution by using the principle of *Simple Forward Checking* with an OR-Parallelism. The call of SFCH is:
`sfch:run({VarNb, DomSZ}, Module, ResCont, ProNb, Result).`

GFCH

This program makes the exhaustive search of solution by using the principle of *Generalized Forward Checking* with an OR-Parallelism. The call of GFCH is:
`gfch:run({VarNb, DomSZ, WZ}, Module, ResCont, ProNb, Result).`

SFCH1

This program searches for a unique solution by using the principle of *Simple Forward Checking* with speculative AND-Parallelism. The call of SFCH1 is:
`sfch1:run({VarNb, DomSZ}, Module, ResCont, ProNb, Result).`

GFCH1

This program searches for a unique solution by using the principle of *Generalized Forward Checking* with speculative AND-Parallelism. The call of GFCH1 is:
`gfch1:run({VarNb, DomSZ, WZ}, Module, ResCont, ProNb, Result).`

1.5.3 The program parameters

The description of the program parameters permits the use of the algorithms, and sets their limitations. If more explanations are required, they may figure in section 1.4.

VarNb

Number of variables. They are encoded from $[0 \ldots (VarNb - 1)]$.

DomSZ

Size of the domains of the variables. Limitation: $DomSZ \leq 32$, because the actual version of the programs uses bit encoding. Domains are encoded from $[0 \ldots (DomSZ - 1)]$.

WZ (only for GFCH and GFCH1)

To decode efficiently the domains of the variables, GFCH and GFCH1 store decoding tables. The size of these tables are in $2^{\frac{DomSZ}{WZ}}$. WZ allows to control the size of the memory to be used. The speed of the algorithms is inversely proportional to this parameter.

Module

Name of the module containing the `constraints/2` predicate, called by the problem solver.

ResCont

Defines the content of the result.

ResCont = 'all' the problem solver outputs a list containing all the solutions of the problem. Each solution is described by a vector, its positions are the variables, and their value the instance. For 6 queens, we obtain:

```
Result = [{4,2,0,5,3,1},{2,5,1,4,0,3},{3,0,4,1,5,2},{1,3,5,0,2,4}]
```

ResCont = 'number' (default parameter) the problem solver outputs a list whose first element is the total number of the solutions found by all the processors; the second element is the list containing the number of solutions found by each single processor. For 6 queens performed in parallel on 2 processors we obtain:

```
Result = [4 [2, 2]]
```

ProNb

Number of processors. According to the load balance principle we adopted, ProNb has to be a power of 2.

Result

Defines the format of the result.

Result is a string The definition of the problem, the time and reduction measures and the result of the algorithm will be written in the file Result. For 6 queens performed on 2 processors with SFCH, the Result file contains:

```
[problem solver,SFCH][problem,queens][size,{6,6}][processors,2][solutions,[4,[2,2]]]
[Constraints,1000][red,4262][time,1000][reductions,5569]
```

The first measures, called `Constraints, red`, give respectively the time and the number of reductions performed to generate the constraints on the processor number 0; the second measures `time, reductions` give the total time and the total number of reductions performed by the algorithm, including the generation of the constraints.

otherwise the result of the algorithm is unified with the variable Result.

1.5.4 Expression of the constraints

The generation of the constraints describing the application is the responsibility of the user. The problem dependent constraints are generated by the `constraints/2` predicate in the `Module` module, as described in Table 1.6 on page 15. The general form of the `constraints` variable is a list of basic constraints $\mathcal{SC}$ $\mathcal{DC}$ expressing the conjunction of its elements; thus constraints may be generated without regard to their order. Their general form is:

constraints({VarNb, DomSZ}, Constraints)
with Constraints = [*StaticConst* ... *DynamicConst* ...]

Static constraints *StaticConst*

The *static constraints* $\mathcal{SC}$ are used during the initialization of the search algorithms, they reduce the domain of value of the variables. Their expression may be paraphrased by: the domain of the variable Var_i is reduced from *DomRes* elements.
There are a maximum of $VarNb$ static constraints. Static constraints are defined by:

$$StaticConst \ = \ [Var_i \ , 0, [Var_i \mid DomRes]]$$

Dynamic constraint *DynamicConst*

The *dynamic constraints* $\mathcal{DC}$ are functions linking an instance of a variable Var_i to other variables, and reducing their domain of values. They may be paraphrased by: when the variable Var_i is instantiated to the value $Inst_i$, then the domain of the variables $Var_j \ldots Var_k$ are reduced from respectively $DomRes_j \ldots DomRes_k$.
The *dynamic constraints* are used by the core of the CSP. As dynamic constraints relate binary relations between variables, and depend on the instance of the variables, they are limited to a maximum of $VarNb^2 \times DomSZ$. Dynamic constraints are defined by:

$$\begin{aligned} DynamicConst \ = \ & [Var_i \ , Inst_i, [Var_j \mid DomRes_j] \ldots [Var_k \mid DomRes_k]] \\ & with \ Var_i \neq Var_j \ \ldots \ Var_i \neq Var_k \end{aligned}$$

A domain restriction, *DomRes*

The domain restriction is the list of the instances to be removed from the domain of the variable to which this list is concatenated. Thus, if one wants to reduce the domain of a variable to an empty set, DomRes contains all the possible instances $DomRes = [0, \ldots (VarNb - 1)]$; conversely, if the domain remains unchanged, DomRes is the empty list $DomRes = [\,]$.

As our `Constraints` are interpreted as the conjunction of its elements, it is better to avoid to generate the constraint having $DomRes = [\,]$, because they do not change the constraints, they make just overwork during the compilation of the constraints. Nevertheless, they do not affect the performances of the Parallel Forward Checking.

A writing facility has been introduced, to reduce the domain of a variable to the empty set. Instead of generating $DomRes = [0, \ldots (VarNb - 1)]$, which is sometimes a long list, we may write $DomRes = ['+']$.

1.5.5 Example of the queen problem

The queen problem exhibits very simple constraints. First of all, there are no static constraints, since all the positions in the domains a *a priori* valid. By running the 3 queens problem, the constraints are the following:

```
queens:constraints({3, 3}, Constraints).
constraints = [[2,0,[1,0,1],[0,0,2]], [2,1,[1,0,1,2],[0,1]],[2,2,[1,1,2],[0,0,2]],
               [1,0,[0,0,1],[2,0,1]], [1,1,[0,0,1,2],[2,0,1,2]], [1,2,[0,1,2],[2,1,2]],
               [0,0,[2,0,2],[1,0,1]], [0,1,[2,1],[1,0,1,2]], [0,2,[2,0,2],[1,1,2]]].
```

Those constraints may be interpreted as: for $x_2 = 0$, then $x_1 \in \{2\}$ and $x_0 \in \{1\}$, and so on. For each element, the line and the diagonals starting form this point, are removed from the domains of validity of the non-determined elements.

1.5.6 Remarks and limitations

For sakes of efficiency, the variables are encoded in numbers ranging from $0 \ldots (VarNb - 1)$, their domains are represented by bit-words, form bit $0 \ldots (DomSZ - 1)$. As a consequence, the size of the domains is limited to 31.

When running the program in parallel, the program adds the needed *static constraints* to perform the work in parallel. Thus the user has just to change the parameter ProNb to run in parallel, keep in mind ProNb is in a power of 2.

The expression of the constraints is a list, interpreted as the conjunction of its elements, thus the following expressions, of dynamic constraints $\mathcal{DC}$ are equivalent:

$$[2, 0, [1, 0, 1], [0, 0, 2]] \Leftrightarrow [2, 0, [1, 1]], [2, 0, [1, 0]], [2, 0, [0, 0, 2]]$$

Beware that the list of the results returned by the Parallel Forward Checking do not follow any order relation mapping the topology of the processors, because the results are gathered with a **merge** operator.

The time measures reported in the run files give following results: The first measures, called **Constraints, red**, give respectively the time and the number of reductions performed to generate the constraints on the processor number 0. Since all the processors generate locally the constraints, the number of reductions given by this measures is proportional to the number of processors. The second measures **time, reductions** give the total time and the total number of reductions performed by the algorithm. This measure waits the end of all the processors tasks.

The overall time and reduction measures are meaningful on all machines, however the partial result of the constraint generation may be false by using simulated parallelism instead of real one.

1.5.7 Error messages

A modest error handler outputs error messages, to prevent against misuse of the programs. This protects user against false results. This error handler outputs messages in a clear language and deadlocks the program. Its current version takes the following cases in account:

$DomSZ \geq 32$ the domain size is too large to be treated by the program, thus the following message will appear:
Error in the domain size DomSz. (0 < DomSz < 32)

$ProNb \neq 2^{something}$ the load balancing has only been foreseen for a power of 2 processors, thus user is warned by:
Error in the processors number ProNb. It should be a power of 2

Unfortunately, the program may make other errors, we encountered one during the experimental study:*memory shortage*, when we tried to compute all the solutions (365596) of 14 queens. Since all the solutions are stored, the machine memory may be upset before the end of the computation.

Chapter 2

Applications, Performances

The *Parallel Forward Checking Problem Solvers* we described in the preceding Chapter may be applied to a large family of problems. These problems are finite domain problems which may be expressed under form of constraints. As Forward Checking is supposed to be one of the most efficient technique [5, 4], we have to give some performance measures to convince (or warn) the potential user.

Four applications have been developed in this Chapter. The first of them is the *Queens* application, described in the section 2.2. The section 2.3 describes the *Zebra problem*, also called the *Five Houses Problem*. Then we studied in Section 2.4 one of the most promising application opening to real-world problems: the *scheduling problem*. In the Section 2.5 we studied the problem of the *mazes*.

Our study started by making a specific algorithm for the exhaustive search of solutions in the *Queen Problem*. It was interesting to compare the efficiency of this specific queen program to the queen problem solved by our *Problem Solver*, presented in Section 2.2. This performance comparison gives an idea of the price paid to generality. These performance comparisons to the related work are presented in Section 2.6.

2.1 Note on performance measures

The performances we give in the following have been measured on the programs written in KL1[15] running on the Multi-PSI/V2[8, 3].

The precision of the measures are approximately of 0.05 seconds; the Multi-PSI has some overheads, and the non-determinacy of the KL1 language adds some uncertainties to the measures. We had to find problems running during several seconds at least, to get reliable results, according to the imprecision of measures.

The total time we give for the run of the algorithms includes everything, the constraints generation, their compilation and the problem solver. The measures we give are, the raw speed, the speedup and the number of reductions. The speedup is calculated by comparing to the same algorithm running on a single processor. This is a fair speedup measure, since the parallelism overhead can be negligible compared to the precision of the measurements.

When analysing the load balance of the processors, we use the number of solutions found by each of them. This measure is not the most precise in terms of the elapsed time, but turns out to be very reliable. It gives the sparseness of the solution space.

2.2 The queen problem

2.2.1 Description of the problem

The *Queen Problem* consists to place n queens on an $n \times n$ board, where no one is attacked by another.

Both kind of experiments, unique and exhaustive solution search, have been carried out on the queen program.

2.2.2 Unique solution search

The queen problem has many solutions, thus it shows strange properties when looking for one solution only. The speculative AND-parallelism we introduced for single solution search brings only minor improvement in the parallel world. We present in the following the results performed for the search of the first solution of 15, 20 and 25 queens.

Raw measures

The Table 2.1 gives the results of SFCH1 and GFCH1 on the unique solution for the 15 queen problem. For GFCH1, we run the program with `WZ=2`, this means that the domains of the variables are decoded in two words when running GFCH1. This allows to reduce the overhead of constructing the decoding tables, On one processor, SFCH1 is almost

program	*number of processors*				
	1	2	4	8	16
SFCH1					
time [s]	1.2	1.09	0.9	1.05	1.16
speedup	1	1.10	1.33	1.14	1.03
Kred	52	98	157	333	606
GFCH1					
time [s]	2.2	1.13	1.09	1.27	1.6
speedup	1	1.95	2.02	1.73	1.38
Kred	75	97	178	358	724

Table 2.1: First solution of 15 queens

two times faster than GFCH1. The speedup we obtained are very slow for SFCH1, one third faster in the best case by using 4 processors. In the parallel world, GFCH1 is approximately as fast as SFCH1, reaching a top 2.02 speedup. The number of reductions are about the same for both algorithms.

The search for the first solution of the 20 queens, in Table 2.2 shows a big performance trade-off between SFCH1 and GFCH1. GFCH1 is 17 times faster than SFCH1, on one processor. SFCH1 shows low speedup with 2 processors,

program	*number of processors*				
	1	2	4	8	16
SFCH1					
time [s]	48.8	44.6	17.1	14.7	8.1
speedup	1	1.1	2.9	3.3	6.0
Kred	2154	4302	3903	5665	6066
GFCH1					
time [s]	2.8	2.6	2.4	2.5	2.9
speedup	1	1.08	1.17	1.12	0.97
Kred	117	226	434	874	1748

Table 2.2: First solution of 20 queens

but achieves a 6.0 speedup with 16 processors. GFCH1 has constant time performances. The difference between the algorithms is explained by their principle, GFCH1 is basically slower, but leads to the good solution, conversely SFCH1 is faster but may search in bad areas of the search space. This is the case here, so speculative parallelism brings speedup.

The study for 25 queens, in Table 2.3 gives as well constant times measures. Here again GFCH1 is faster than SFCH1, by a ratio of 3. However this trade-off is due to another reason than before, it is due to the size of the search space.

The run in paralle does not bring any improvement, the time remains strictly constant. We can notice the evolution

program	*number of processors*				
	1	2	4	8	16
SFCH1					
time [s]	14.4	14.4	14.3	14.4	14.6
speedup	1	1	1.01	1	0.99
Kred	697	1280	2770	5620	11286
GFCH1					
time [s]	4.9	4.8	4.7	4.8	4.9
speedup	1	1.02	1.04	1.02	1
Kred	214	422	840	1667	3298

Table 2.3: First solution of 25 queens

of the number of reductions, growing proportionally to the number of processors. It shows the good behaviour of the algorithms, where all processors work until one solution is detected,

Raw speed, speedup and reductions

With a small search space (10 and 15 queens), SFCH1 is faster than GFCH1, because of its simplicity. Conversely, with an increased search space, GFCH1 takes advantage over two phenomenons, it focuses better the searches towards the solution (as seen in 20 queens), and is much faster in large search spaces because of its early pruning especially (see 25 queens). Some problems seem to be particularly unfavorable to SFCH1, because it begins a search in a bad subpart of the search-space. In this cases *speculative AND-parallelism* brings good, maybe super-linear speedups. But is remains speculative, thus in other case we win nothing.
Speculative AND-parallelism did not bring great results in the queen problem. This was foreseen, since the search space of the queen problem has many solutions, the speculative work does not find magical solutions. In some cases, speculative work may bring speedup or slowdown for SFCH1, but GFCH1 has approximately constant performances.

2.2.3 Exhaustive solution search

We run the *Parallel Forward Checking* exhaustive search with 8, 10, 12, 13 queens. Thus the OR-parallelism of the search can be studied. In the first subsection we give the raw measures with some comments, than we study the load balancing of the processors.

Later on, in section 2.6 the results of the *Parallel Forward Checking* we present here will be compared to the specific implementations of the queen problem. So we may measure the trade-off due to the encapsulation of the *Problem Solvers*. The conclusions specific to the queen problem, are shown as well in section 2.6.

Raw measures

Table 2.4 gives the results of the 8 queen problem. These results should be considered with care, because the elapsed time to calculate the 92 solutions is very small compared to the precision of the measures on the Multi-PSI. SFCH is

program	*number of processors*				
	1	2	4	8	16
SFCH					
time [s]	0.59	0.53	0.47	0.56	0.66
speedup	1	1.1	1.25	1.05	0.88
Kred	17	23	35	60	112
GFCH					
time [s]	1.2	0.97	0.75	0.59	0.72
speedup	1	1.23	1.6	2.02	1.67
Kred	26	34	50	81	147

Table 2.4: FCH Application on 8 queens (92 solutions)

faster than GFCH, but the latter takes a better advantage of the parallelism, especially when dealing with 8 processors. In terms of reductions, we note a dramatic increase with the number of processors. This is due to our constraint generation performed in parallel on all the processors. Speed of the parallel processors cannot be improved to show great jumps since the generation of the constraints, which takes 0.18 ms is done sequentially on all processors.

More reliable results can be seen by running 10 queens, as reported in Table 2.5. Here the elapsed time are better suited to the precision of the measures. GFCH is almost 3 times slower than SFCH, on one processor. Again its

program	*number of processors*				
	1	2	4	8	16
SFCH					
time [s]	5.54	3.32	1.93	1.28	1.25
speedup	1	1.67	2.87	4.32	4.43
Kred	194	205	230	279	383
GFCH					
time [s]	14.6	7.66	4.36	2.88	2.67
speedup	1	1.9	3.34	5.07	5.47
Kred	314	336	370	448	612

Table 2.5: FCH Application on 10 queens (724 solutions)

speedups are higher than SFCH, attaining more than 5 with 16 processors. GFCH makes approximately two times more reductions than SFCH. A graphic representation of this results is drawn in Figure 2.9 on page 35.

The Table 2.6 gives the results of the run of 12 queens. We may begin to trust in these results since the overheads are negligible compared to the tasks performed. The algorithm works very well on two processors, achieving super-linear

program	*number of processors*				
	1	2	4	8	16
SFCH					
time [s]	127.7	61.2	32.6	20.2	13.8
speedup	1	2.08	3.92	6.32	9.25
Kred	4380	4400	4442	4528	4719
GFCH					
time [s]	325.2	163.1	84.1	52.3	37.1
speedup	1	1.99	3.86	6.21	8.76
Kred	6526	6627	6684	7000	7498

Table 2.6: FCH Application on 12 queens (14200 solutions)

speedup for SFCH and 1.99 speedup for GFCH. The results on 4 processors are as well impressive. At the very end, SFCH gets higher speedups than GFCH. Again SFCH is about 2.5 times faster than GFCH. A graphic representation of this results is drawn in Figure 2.10 on page 36.

Our last experiment on the queens is the 13 queens, reported in Table 2.7. The gap between SFCH and GFCH is increasing. Here we notice a super-linear speed-up on GFCH dealing with 2 processors. The final speedup obtained with 16 processors is about 10. We notice as well the evolution of the reductions, which are almost constant, whatever the number of processors. A graphic representation of this results is drawn in Figure 2.11 on page 37.

Raw speed, for single solutions

Having the raw measures, it is interesting to transform them to see the time needed to calculate one solution. We make the ratio $\frac{raw\ time}{number\ of\ solutions}$ in Table 2.8 for the SFCH. All The measures are expressed in milli-seconds. The solution search on a single processor increases with the number of queens, nevertheless, this increase is inferior to linear, thanks to the bit encoding.

program	*number of processors*				
	1	2	4	8	16
SFCH					
time [s]	690	366	222	117.4	71.2
speedup	1	1.88	3.11	5.89	9.69
Mred	23.6	23.7	23.7	23.8	24.1
GFCH					
time [s]	1941	941	516	301	182
speedup	1	2.06	3.76	6.44	10.66
Mred	34.5	34.7	34.7	35.5	36.5

Table 2.7: FCH Application on 13 queens (73712 solutions)

program	*number of processors*				
	1	2	4	8	16
8 Queens time [ms]	6.4	5.8	5.1	6.1	7.2
10 Queens time [ms]	7.6	4.6	2.7	1.8	1.8
12 Queens time [ms]	9.0	4.3	2.3	1.4	1.0
13 Queens time [ms]	9.4	5.0	3.0	1.6	1.0

Table 2.8: Average time for 1 solution with SFCH

A deeper study of the properties of the queen problem follows in section 2.6. The reader will find on page 39 the equivalent Table 2.22, for the specific queen algorithm.

Load balancing

As said in the introduction to parallelism, the key of the success lies in an ideal load balance, thus it is necessary to study the results obtained by the simple manner we introduced. To do so, we stored the number of solutions found by each processor. Even if this is not a proof, it seems to be a good presumption of the work done by a processor. Remind that the load balance is the same for SFCH and GFCH.

The load balance for 2 and 4 processors share the same properties. If the number of queens is even, the load balance is optimal, see Figure 2.1. As a consequence we obtain excellent results in those cases, and achieved several times super-linear speedup (in Table 2.6, 2.7, 2.18, 2.20).

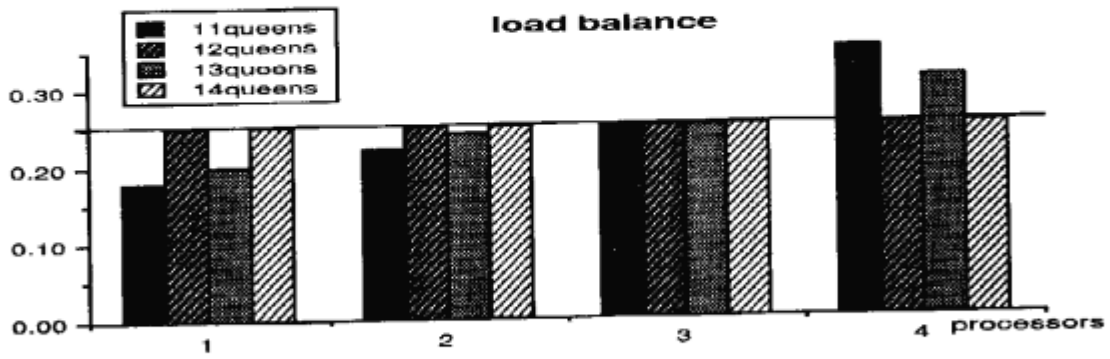

Figure 2.1: Load balance for 4 processors

Unfortunately, when 8 processors are used, the "magic properties" of the load balance are fading, some processors have much more work to do then others. The load balance is not catastrophic, but gives some clues towards the explanation of a proportionally lower efficiency with 8 processors than with 2 or 4. Nevertheless, the load balance is improved as the number of queens increases, see Figure 2.2. In the same manner, Figure 2.3 shows the load balance

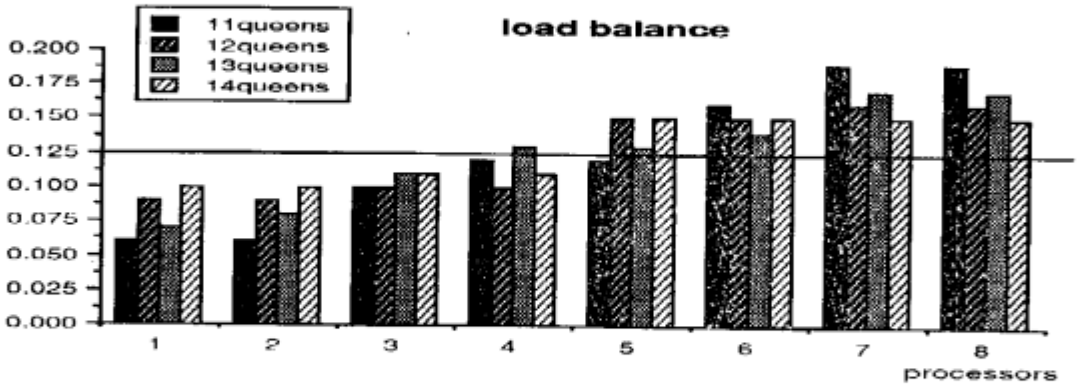

Figure 2.2: Load balance for 8 processors

of 16 processors. Here too, we notice a better load balance with an even number of queens. The range of the load distribution is decreasing as the size of the problem increases. For the even number of queens, 8 processors are near on the optimum load distribution.

Figure 2.3: Load balance for 16 processors

2.3 The zebra problem

2.3.1 Description of the problem

This is a well known problem, due to Lewis Caroll.

Five men with different nationalities live in the five first houses of a street. They practice five distinct professions, and each of them has a favorite animal, a beloved drink, all of them being different. The five houses are painted in different colors. The following facts are known:

The English man lives in a red house

The Spaniard owns a dog.

The Japanese is a painter

The Italian drinks tea

The Norvegian lives in the first house on the left

The owner of the green house drinks coffee

The green house is on the right of the white one

The sculptor breeds snails

The diplomat lives in the yellow house

Milk is drunk in the middle house

The Norvegian's house in next to the blue one

The violinist drinks fruit juice

The fox is in the house next to that of the doctor

The horse is in the house next to that of the diplomat

The problem is to identify who owns the zebra and who drinks water.

We implemented 2 versions of this problem. In the first, `Zebra1`, the street where this people live is straight, the solution of the problem is unique. In the second version, `Zebra`, we consider the case of a circular street, thus the problem has 21 solutions, the constraints are looser.

The subsections are organized in the following way, at first we run the program with the straight street. As the solution is unique, we use the programs searching for single solutions. Then we turn our interest to the circular street. This problem requires a little more work to be solved. We may have interesting conclusions on the parallelism, since both, *Speculative AND-parallelism* and *OR-parallelism* are used.

2.3.2 The straight street: Zebra1

Raw measures

Table 2.9 reports the results of SFCH1 and GFCH1 applied to the Zebra problem. As the problem is well constrained, it is solved is approximately 0.5 seconds. When running this problem on several processors, by using the speculative AND-parallelism, we got a good surprise in achieving some speedup. The best results are obtained with 2 processors, respectively for SFCH1 and GFCH1. Even these low speedups were unexpected since most of the papers in parallel literature pretend that the Zebra problem has no inherent parallelism.

2.3.3 The circular street: Zebra

When considering all the folks of the zebra problem inhabiting around a square, some more solutions of the problem are available because the house 1 and 5 are neighbors.

program	*number of processors*				
	1	2	4	8	16
SFCH1					
time [s]	0.48	0.40	0.52	0.58	0.94
speedup	1	1.2	0.92	0.82	0.51
Kred	16	26	61	113	232
GFCH1					
time [s]	0.67	0.50	0.59	0.6	0.93
speedup	1	1.34	1.14	1.12	0.72
Kred	19	32	63	119	239

Table 2.9: FCH Application of speculative AND-parallelism to Zebra1

Raw measures

As we are looking for all solutions, we run SFCH and GFCH. They find 21 solution of the problem. Parallelism is done in Or-parallelism. This problem gets some tiny speedups, less than 2 in the best case. Note the dramatic increase of

program	*number of processors*				
	1	2	4	8	16
SFCH					
time [s]	0.69	0.59	0.62	0.57	0.78
speedup	1	1.17	1.11	1.21	0.88
Kred	22	36	65	122	236
GFCH					
time [s]	1.16	0.89	0.68	0.62	0.81
speedup	1	1.3	1.71	1.87	1.43
Kred	29	44	73	132	250

Table 2.10: FCH Application of OR-parallelism to Zebra (21 solutions)

the number of reductions, when the number of processors increases.

2.3.4 Interpretation of the results

Despite the low speedups, these results in parallel environment were unexpected. We were more thinking about some slow down in parallel. The choices we made for the parallel implementation explain these results. As we work in static load balancing, spreading small data, a negligible overhead is paid to the parallel implementation. On the other hand, this requires each processor to generate locally the constraints of the problem. This constraint generation takes approximately 0.25 seconds and 8600 reductions. So this explains the increase of the reductions performed in the parallel environment, but as well it sets the maximum parallelism of the zebra problem to a speedup factor of 2.7 for SFCH and 4.64 for GFCH, according to Amdahl's law.

2.4 The scheduling

2.4.1 Description of the problem

Scheduling is an NP-complete problem. Hence in most of the cases the scheduling is to realize a task in a finite time, it is an excellent application for our problem solvers. If the domain of search is large, parallel implementation may attain good speedups.

But more than a toy problem, scheduling is an application needed in various fields of sciences, in the industry and everyday life. Its application in VLSI is one of the well known examples.

Thus, we designed more than just an application program making scheduling on an example. We designed a language to express the scheduling constraints under form of a graph of temporal dependencies. Our program is restricted to *hamiltonian graphs*. It proceeds as follows:

construction of the *predecessor*, *successor* graphs.

construction of the *As Soon As Possible* and *As Late As Possible* scheduling.

Definition of the domains of the variables.

Generation of the constraints for the *Parallel Forward Checking*.

In the following we show some scheduling applications. The first of them is extracted from the field of Design Automation [10, 9]. It may be paraphrased by the grammar:

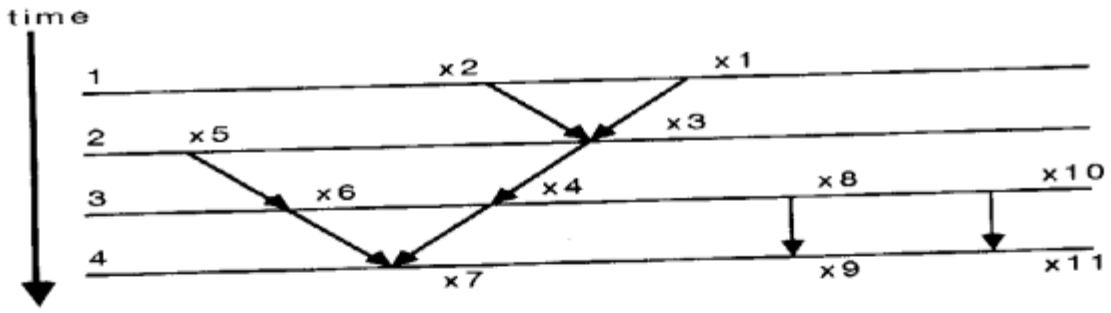

Figure 2.4: Scheduling used for Design Automation

$$
\begin{aligned}
x_1 &\rightarrow x_3 \\
x_2 &\rightarrow x_3 \\
x_3 &\rightarrow x_4 \\
x_4 &\rightarrow x_7 \\
x_5 &\rightarrow x_6 \\
x_6 &\rightarrow x_7 \\
x_8 &\rightarrow x_9 \\
x_{10} &\rightarrow x_{11}
\end{aligned}
$$

The language we made takes this grammar as starting point of the algorithm. This design example is very small, having only 11 variables and a critical time of 4 time steps. This example is composed of 3 independent subgraphs.

Then, we treated a more complicated example representing a graph. This example gives an idea of the complexity of the scheduling problems allowed by our language. This example deals with 18 variables, and may be solved in 9 time steps, but its definition graph has 15 time steps. The reader may easily construct the grammar describing this example, by analogy to the preceding one.

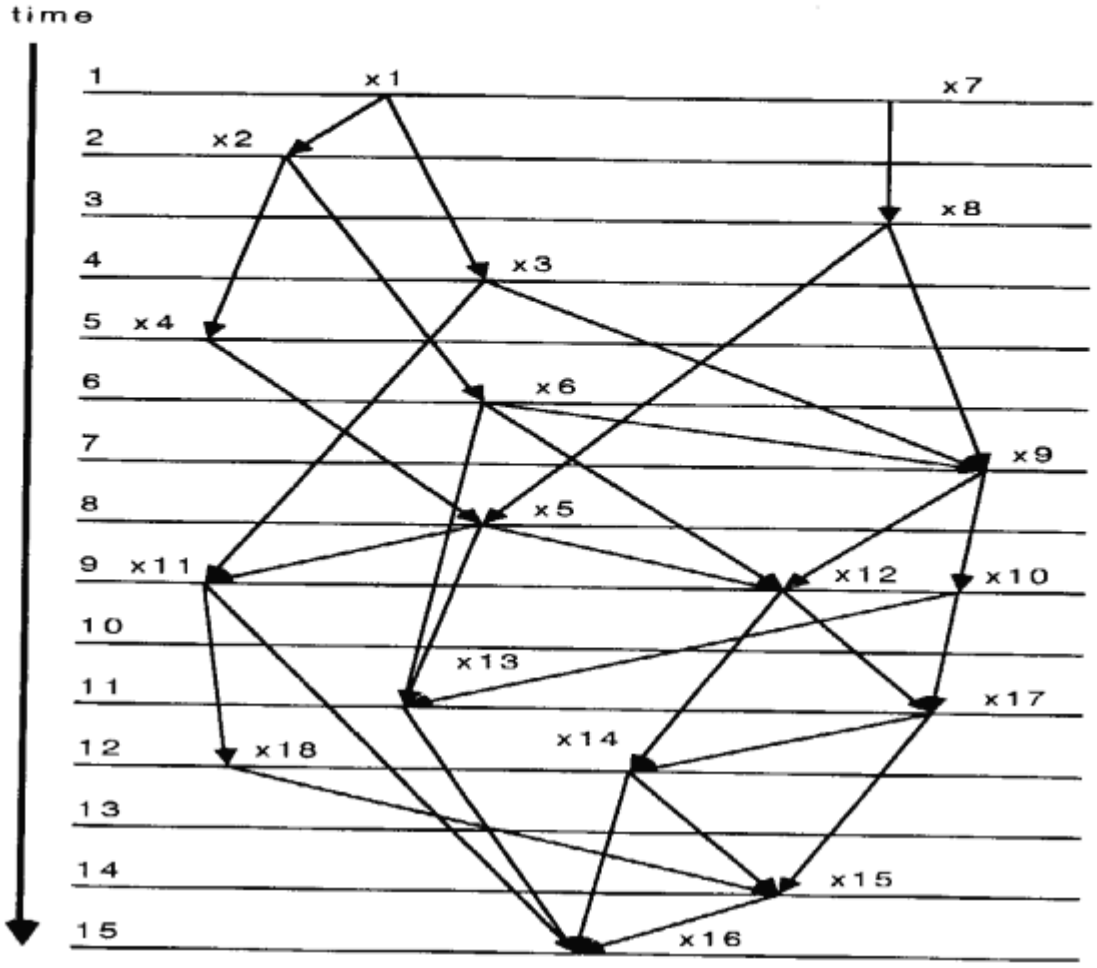

Figure 2.5: Scheduling in a graph

2.4.2 Unique solution search

When running unique solution search in shceduling, results are found in very short time, 0.2 or 0.3 seconds. Thus de do not report the results here. Sequential implementation is the fastest, then the search time increases slightly, maybe 10%.

2.4.3 Exhaustive solution search

Raw measures

When solving the Automatic Data Path for high level design, the results for the critical time are found in 0.3 seconds. Thus we give looser constraints, providing a bigger search space, some speedup are achived in parallel, see Table 2.11. In 6 time-steps, the algorithm finds 53775 solutions. SFCH is between 2 and 3 times faster than GFCH. The speedups are a little better for GFCH, approximately 5%.

If we apply our algorithms to solve the scheduling of the graph, we obtain again the solution in 0.5 seconds and 0.6 seconds, respectively for SFCH and GFCH; provided the critical time, 9 time steps. When releasing the constraints to 11 time steps, more solutions are available, as shown in Table 2.12. GFCH is two times slower than SFCH, in this example. The speedups are very low, attaining approximately a three-fold factor for 16 processors.

Raw speed, speedup, reductions

Scheduling does not show impressive speedups. The GFCH is very slow compared to the SFCH, this may be explained by the search space, containing many variables (18 for the graph) but its depth is just 11 in our case. Another reason

program	*number of processors*				
	1	2	4	8	16
SFCH					
time [s]	28.5	17.6	11.8	7.3	5.1
speedup	1	1.6	2.4	3.9	5.6
Kred	907	909	914	924	945
GFCH					
time [s]	65.4	36.2	24.1	14.4	10.5
speedup	1	1.8	2.7	4.5	6.2
Kred	1640	1624	1542	1541	1552

Table 2.11: All solutions of the automated Desing problem in 6 time-steps(53775 solutions)

program	*number of processors*				
	1	2	4	8	16
SFCH					
time [s]	27	24.8	21.3	12.7	9.2
speedup	1	1.09	1.27	2.13	2.93
Kred	866	908	993	1163	1501
GFCH					
time [s]	60.2	56.9	45.7	26.7	18.6
speedup	1	1.06	1.32	2.25	3.24
Kred	1588	1631	1683	1860	2222

Table 2.12: All solutions of scheduling in 11 time-steps (46926 solutions)

for the efficiency of the SFCH is the nature of the problem. Since it is described by a graph, the constraints are expressed and used in an efficient frame, unnecessary branches are pruned early.

Load balancing

The study of the load balancing, shown in Figure 2.6, explains the poor speedup obtained by the scheduling program. The work is shared in an unfair manner between the processors, and the speedup cannot be higher as long as the load balance is not improved. The figure represents the load balance for 2 and 16 processors, on the graph example. For 2 processors, we see that one of them is really lazy, performing only 9% of the work. With 16 processors, this figure is slightly better, but the most buzy of the processors makes still one third of the work. There is an obvious correlation between load balance and the speedup found in Table 2.12.

2.4.4 Interpretation of the results

The scheduling program provides a convenient environment to solve every day problems, if they may be solved in less than 32 time steps.

SFCH seems to be more efficient than GFCH in scheduling problems, by a factor of two. Since constraints are expressed under form of a graph, they seem to be well used in SFCH.

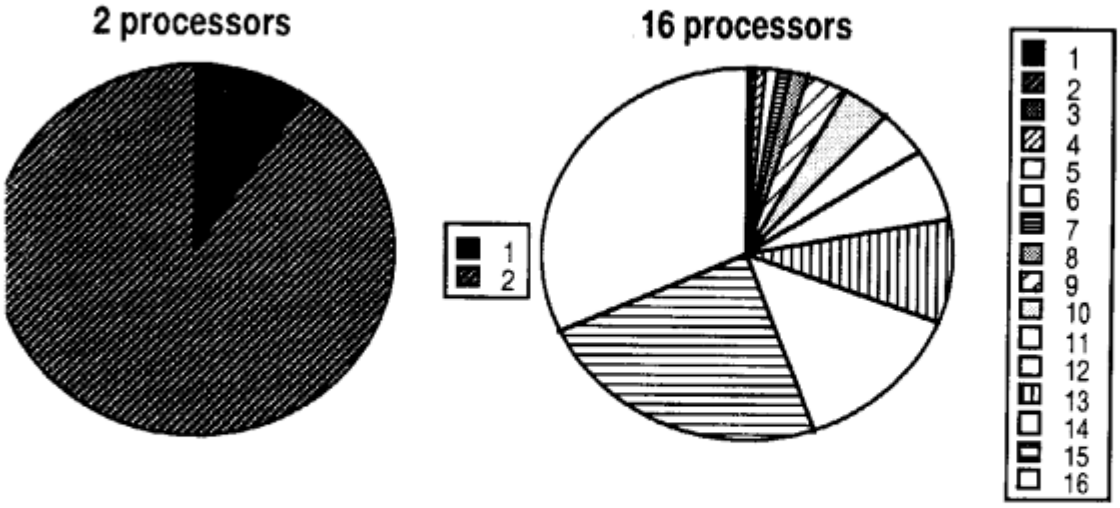

Figure 2.6: Load balance in scheduling

2.5 The maze

Mazes or labyrinths are finite space search problems. However, the labyrinths human beings are used to solve present quite often high constraints, typically a small search space, but deep backtracks. These mazes challenge human beings in finding the solution in the shortest time. Hence search spaces are small, machines are not challenged by such mazes, being rather trivial. However, if the constraints of the mazes get looser, search spaces grow and labyrinths become interesting search problems for machines (but trivial ones for human being). This explains the surprising structures of labyrinth we use for the benchmarks.

2.5.1 Description of the problem

We defined three different mazes to make our experiments. All of them have a parameter which is the length of the maze. Their proportions and shapes remain constant, whatever the length parameter. They are represented in Figure 2.7. Imagine we control a robot entering these mazes. It moves form the left to the right, the possible moves

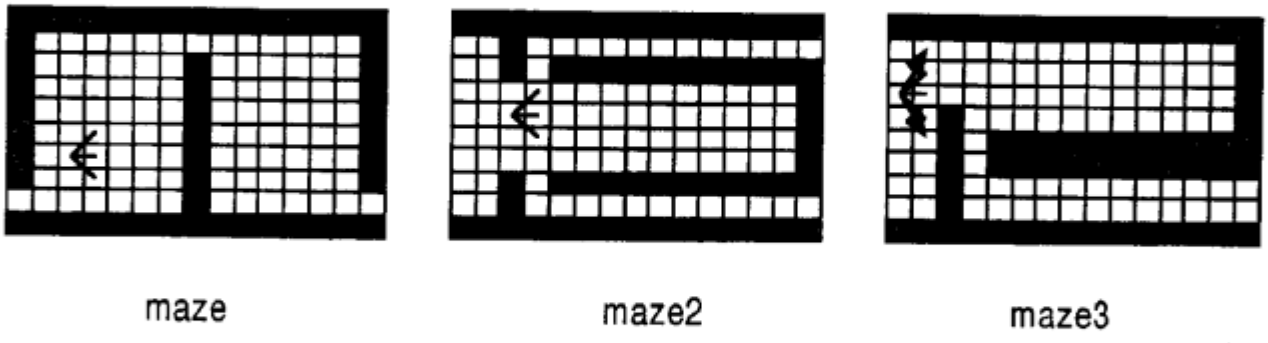

Figure 2.7: Definition of the mazes

being the 3 adjacent cases for *maze* and *maze2* and the 5 adjacent for *maze3*. The black cases represent the forbidden cases in the maze.

2.5.2 Unique solution search

Raw measures

Tabe 2.13 represents the resuls for a sole result of maze(31). The results are almost constrant, whatever the problem solver or the number of processors. As all the solutions of the maze are in the same geographic area of the maze,

program	*number of processors*				
	1	2	4	8	16
SFCH1					
time [s]	3.0	2.9	3.0	2.9	3.2
speedup	1	1.03	1	1.03	0.94
Kred	212	423	841	1659	3295
GFCH1					
time [s]	3.3	3.0	3.1	3.3	3.7
speedup	1	1.1	1.06	1	0.89
Kred	215	426	853	1709	3425

Table 2.13: Unique solution of maze(31)

speculative AND-parallelism cannot bring interesting speedups, the constraints of the problem lead efficiently one processor towards the solution. The number of reductions is proportional to the number of processors. This shows again that communication time may be neglected, tanks to static load balancing.

In maze3, the behaviour of SFCH1 and GFCH1 are completely different. GFCH1 leads efficiently to the solution, but SFCH1 is lost in the maze, looking of all the solutions. The results shown in Table 2.14 show well the difference between both problem solvers, GFCH1 being 75 times faster than SFCH1 on this problem, with one processor. The number of reductions performed by the algorithms shows the trade-off. As the number of processors increases, GFCH1

program	*number of processors*				
	1	2	4	8	16
SFCH1					
time [s]	28.9	25.2	13.9	6.4	3.0
speedup	1	1.15	2.08	4.52	9.63
Kred	1019	2017	2166	1951	1621
GFCH1					
time [s]	0.38	0.40	0.48	0.65	1.0
speedup	1	0.95	0.79	0.58	0.38
Kred	10	19	49	113	267

Table 2.14: Unique solution of maze3(13)

gets slower whereas SFCH1 speeds up. There is no good explanation available for the slow down of GFCH1. SFCH1 speeds up because of luck, the search space being smaller, there are more chances to find the solution early.

2.5.3 Exhaustive solution search

Raw measures

Again, we use maze to perform the exhaustive search. The results in Table 2.15 show the good performance of SFCH, which is 2.5 times faster than GFCH. The overal speedups by both algorithms is 6. We may notice that GFCH makes approximately 2 times more reductions than SFCH. The price of GFCH's overhead is very high is this case, and leads to the poor results.

Conversely, when running the problem solvers on maze2(17), (see Table 2.16), GFCH is approximately 1500 times faster than SFCH. SFCH is trapped in maze2, and searches in the whole search space, but finally, fails to find the pathes in the wall. GFCH begins with the most constraint variable, this means with the wall, then propagates the results to the starting point. Thus most of the search space is pruned in the early stages. Very interestingly, we note

program	*number of processors*				
	1	2	4	8	16
SFCH					
time [s]	241	140	90	64	41.3
speedup	1	1.7	2.7	3.8	5.8
Kred	8.6	8.8	9.2	10.2	11.9
GFCH					
time [s]	636	352	227	152	106
speedup	1	1.8	2.3	4.2	6
Kred	15.5	17.4	19.2	22	27.3

Table 2.15: All solutions of maze(31) (444315 solutions)

program	*number of processors*				
	1	2	4	8	16
SFCH					
time [s]	1110	565	390	274	179
speedup	1	1.96	2.85	4.05	6.2
Mred	36.6	36.7	36.7	36.8	37
GFCH					
time [s]	0.70	0.72	0.83	1.1	1.53
speedup	1	0.97	0.84	0.64	0.45
Kred	35	69	138	275	552

Table 2.16: All solutions of maze2(17) (18 solutions)

again a slowdow of GFCH when more processors are used. The results show coherency to the ones of *maze3* in the first solution search.

Raw speed, speedup, reductions

The parallel implementation showed very poor performances, as a consequence of the load balancing. The current version of the load balancing shares the search spaces between processors by halfing the first variables of the search space. Obviously this leads to bad results when the domains of these variables are small. SFCH is trapped in maze2, and searches in the whole search space, from left to right, but finally, fails to find the pathes by crashing into wall. GFCH begins with the most constraint variable, this means with the wall at the right, then propagates the results to the starting point. Thus most of the search space is pruned in the early stages.
Labyrinths are convenient illustrations to show the properties of SFCH and GFCH. The lesson we take from these experiments is that efficiency of the algorithms depends on the nature of the constraints expressed by the application.

With simple and regular constraints, SFCH is faster, but in some cases it is more important to take dynamically advantage of the available information and to propagate the "best" constraints, then GFCH becomes faster.

The speedups attain a 6 factor for SFCH and GFCH in *maze*. we note impressive slowdown for GFCH1 and GFCH when running on *maze3* and *maze2*. There is no explanation for them for the moment.

Load balancing

We studied the load balance on the *maze* problem. It is represented in Figure 2.8. The load balance for 2 processors is near form optimal whereas the one for 16 processors has 2 processors taking approximately one sixth of the work each. This load balance is not a desperate case, but sets strong limitations on the speedup with more than two processors.

2.5.4 Interpretation of the results

The difference of principle between SFCH and GFCH may lead to terrible performance trade-off. GFCH seems to be a safe solution to problems, often slower than SFCH, but in reasonable factors, less than an order of magnitude. Conversely, SFCH could be seen as a brilliant algorithm, often outperforming GFCH, but sometimes trapped. In such

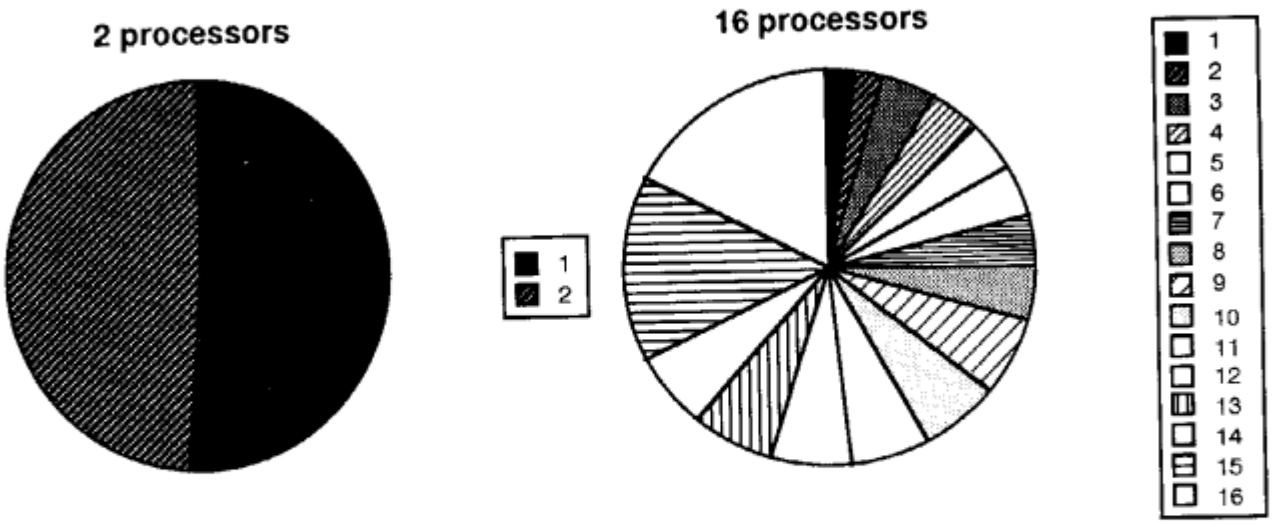

Figure 2.8: Load balance in maze

cases the performances are really terrible.

Once more we could explain the speedup performances by the load balance of the algorithm. One mystery remains with the surprising behaviour of the GFCH and GFCH1 in parallel on *maze* and *maze3*.

2.6 Related work

In this section we present a comparison between the results obtained by the Problem-solvers (reported in section 2.2 and the results of the specialized queen programs. Three specialized queen programs have been implemented, the *Q_SFCH*[1] (for *Queens Simple Forward Checking*) and two versions of the *Q_GFCH* (*Queens Generalized Forward Checking*). These specialized programs do exhaustive search of the solutions. The difference between the latter ones is in the size of the stored data. *Q_GFCH* stores two decoding tables having 2^n elements each, n being the number of queens. As this size is not reasonable, we implemented the other version, *Q_GFCHh* which stores two tables having $2^{\frac{n}{2}}$ elements. To perform the decoding of the number, the *Q_GFCHh* algorithm has to do it in two times by using bit shifting.

Three subsections follow, the first presents the raw measures of the algorithms, the second gives the evolution of the speedup and the number of reductions and raw speed according to the number of queens, and the third subsection studies the load balancing.

2.6.1 Raw measures

Most of the papers give the time measures for the 8 queens problem. We start with the 10 queens because the 8 queens are done in approximately 0.4 seconds on one processor, the measures facilities provided by the Multi-PSI are not precise enough to relate the 8-queens problem, and results would be odd.

10 queens

program	*number of processors*				
	1	2	4	8	16
Q_SFCH					
time [s]	4.85	2.6	1.4	0.97	0.86
speedup	1	1.86	3.34	5	5.63
Kred	183	184	185	192	208
Q_GFCH					
time [s]	6.6	3.4	1.9	1.2	1.1
speedup	1	1.94	3.47	5.5	6
Kred	161	166	178	204	271
Q_GFCHh					
time [s]	10.2	5.2	2.7	1.7	1.4
speedup	1	1.96	3.78	6	7.29
Kred	206	206	205	206	226

Table 2.17: Queen Program: 10 queens (724 solutions)

Table 2.17 gives the raw speed measures in seconds, the speedup and finally the number of reductions, in Kilo or Mega reductions. The most efficient program is *Q_SFCH*, whatever the number of processors. The *Q_GFCH* is approximately 40% slower, the *Q_GFCHh* is two times slower. The speedup of the *Q_GFCHh* is better than the ones from the other programs, see Figure 2.9. The number of reductions of the *Q_GFCH* is the lowest when running on one processor, it increases proportionally with the number of processors. This increase is explained by the duplication of the tables on each of the processors. This remark is also valid for *Q_SFCH* and *Q_GFCHh*, having respectively 10% and 20% more reductions.

12 queens

Table 2.18 gives the detailed results of the run. Compared to the previous results, *Q_GFCH* is gaining some efficiency, compared to *Q_SFCH*, but is still slower from 10% approximately. *Q_GFCHh* is also improved, as it's principle is the same as *Q_GFCH*, but is still far from *Q_SFCH*, in time performance, see Figure 2.10. The maximum speedup reached a tenfold factor for 16 processors. Generally speaking, the speedup for 2 and 4 processors are excellent, and two of them are super linear (they are emphasized by a rectangle). The number of reductions shows small variations with the number of processors, *Q_GFCH* makes 40% less reductions than *Q_SFCH* while *Q_GFCHh* makes 10% less.

[1]The sequential version of this program has been written by Takashi Chikayama

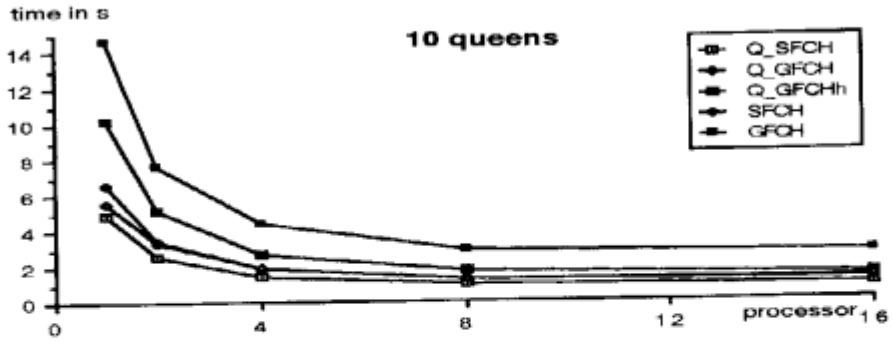

Figure 2.9: 10 queens (724 solutions)

program	*number of processors*				
	1	2	4	8	16
Q_SFCH					
time [s]	115.5	56.2	30.3	18.8	12.4
speedup	1	2.05	3.81	6.14	9.31
Kred	4361	4363	4366	4375	4414
Q_GFCH					
time [s]	135.4	71.6	33.1	20.5	13.4
speedup	1	1.89	4.09	6.6	10.1
Kred	3124	3180	3112	3180	3382
Q_GFCHh					
time [s]	216.6	110.8	54.4	34	20.5
speedup	1	1.95	3.97	6.37	10.57
Kred	4099	4142	3982	3937	3932

Table 2.18: Queen Program: 12 queens (14200 solutions)

13 queens

The results of the 13 queens are shown in Table 2.19. *Q_GFCH* is improving its results, but is still slower than the *Q_SFCH*, even with 16 processors. *Q_GFCHh* is very slow on 1 processors, and gets better speedup, but is still 1.5 times slower with 16 processors. Figure 2.11 shows the results. The speedup are quite bad for 2 and 4 processors, if compared to the ones of 12 queens. As we show it later, this comes from the load balancing: the domain of values are split into 2, so the load balance works well for even number, and less for odd ones. Nevertheless, the speedup with 16 processors seems not to be affected by these considerations, and reaches a top of 10.7. The number of reductions of *Q_SFCH* is increasing with the number of processors, conversely the number of reductions of *Q_GFCH* and *Q_GFCHh* are decreasing with the number of processors. This new tendency may be explained by a better use of the constraints by the *Q_GFCH*, the initial problem on each processor gets higher constraints in parallel, thus *Q_GFCH* turns out to be more efficient.

14 queens

The run of the 14 queens are given in Table 2.20. For the first time, *Q_GFCH* becomes more efficient than *Q_SFCH*, this with 16 processors only. But the difference is too small to be significant, since measures are relying on the non-determinacy of KL1. Speedups also become interesting, reaching 11.8. Again, notice the super linear speedup for 2 and 4 processors, where the algorithm is respectively 2.03 and 4.21 times faster than with one processor. Figure 2.12 represents the raw time performance of the algorithms. The trends or the reductions get more obvious, in the best case, *Q_GFCH* makes just 60% of the reductions of *Q_SFCH*, but they need the same time. The reductions of *Q_GFCH* are more complicated than the ones from *Q_SFCH*, furthermore, *Q_GFCH* carries bulky data.

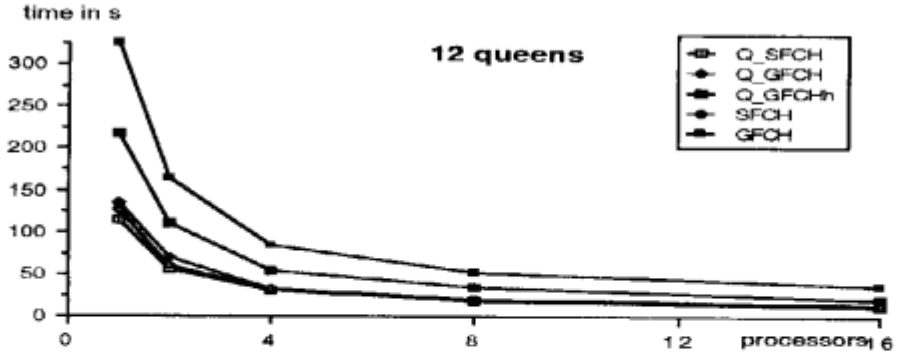

Figure 2.10: 12 queens (14200 solutions)

program	*number of processors*				
	1	2	4	8	16
Q_SFCH					
time [s]	616	332	186	107	65
speedup	1	1.86	3.31	5.76	9.48
Kred	23641	23643	23646	23658	23713
Q_GFCH					
time [s]	718	404	217	116	67
speedup	1	1.78	3.31	6.19	10.72
Kred	16424	16408	15865	15773	16135
Q_GFCHh					
time [s]	1129	632	333	179	98
speedup	1	1.79	3.39	6.31	11.52
Kred	21544	21448	20595	20202	20129

Table 2.19: Queen Program: 13 queens (73712 solutions)

15 and 16 queens

The 15 and 16 queens problems have been run on 16 processors only, to compare Q_SFCH and Q_GFCH. Unfortunately, we cannot see the speedup in this case. The results are summarized in Table 2.21. This confirms the tendency we noted earlier, Q_GFCH is getting faster with larger search spaces. The number of reductions gives an image of the trade-off between both programs. Surprisingly the difference between both programs is stable, about 15% is speed and 70% in reductions.

2.6.2 Raw speed, speedup and reductions

It is interesting to analyze the results in a perpendicular plan, to see the evolutions of the performances varying with the number of queens.

Raw speed, for single solutions

Having the raw measures, it is interesting to transform them to see the time needed to calculate one solution. We make the ratio $\frac{raw\ time}{number\ of\ solutions}$ in Table 2.22. We consided always the best solution between Q_SFCH and Q_GFCH. All The measures are expressed in milli-seconds. The ones with a $\star$ have been recorded with Q_GFCH. In the sequential version, the time to find a solution increases with the size of the problem. In the parallel world this does not held, and the time decreases as the size of the problem increases. The parallelism gains efficiency with the size of the problems.

The speedups

The speedup for 2 and 4 processors are near on the ideal one, then the speedup are fadding, according to the load balancing studied in section 2.2. The Figure 2.1 in page 23, Figure 2.2 in page 24, Figure 2.3 in page 24, give direct

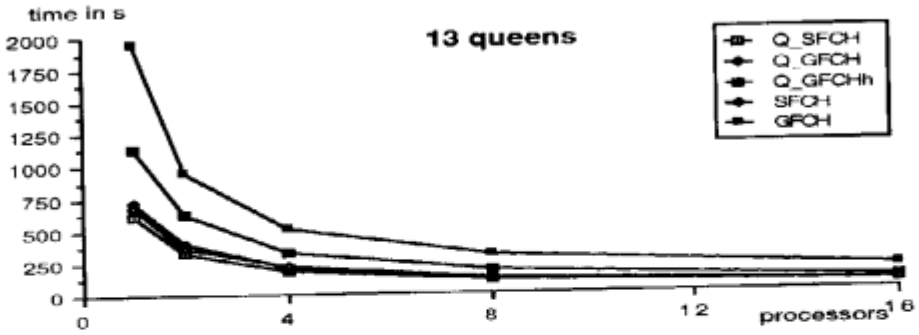

Figure 2.11: 13 queens (73712 solutions)

program	*number of processors*				
	1	2	4	8	16
Q_SFCH					
time [s]	3676	1806	943	551	348
speedup	1	2.03	3.89	6.67	10.56
Mred	136	136	136	136	136
Q_GFCH					
time [s]	4066	2005	966	565	344
speedup	1	2.03	4.21	7.2	11.82
Mred	93	91	86	87	85
Q_GFCHh					
time [s]	6284	3102	1489	868	526
speedup	1	2.03	4.22	7.24	11.95
Mred	120	119	112	111	109

Table 2.20: Queen Program: 14 queens (365596 solutions)

explanation of the speedups.

It is worth to note the good speedups of the even number of queens, compared to the odd ones. Especially 11 queens shows terrible results. Our load balance is well suited to search spaces in even size, because the sub domains used for the parallel processing are of equal size. This problem fades with large search spaces.

The evolution of the number of reductions, is very important to predict the performances of the algorithms and their trends. Figure 2.13 gives this evolutions, function of the number of queens. As the number of reductions increases exponentially, we represented it with logarithm scale. The kkqueens algorithm represented in the figure, is one of the classical queen program described in [14], we give it to set a reference point for the forward checking. With the augmentation of the number of queens, the Q_GFCH makes less reductions than Q_SFCH, almost half as much for 15 and 16 queens, but the time performance are just 10% better. Figure 2.13 gives the number of reductions when using 16 processors. The Figure 2.14 gives the evolution of the time performance with the number of queens. Conversely to Figure 2.13, Q_SFCH shows better performances than Q_GFCHh, despite a higher number of reductions. Notice as well the effect of the bad load balance with 11 processors, leading to catastrophic results for Q_GFCHh.

2.6.3 Interpretation of the results

The previous sections detailed the results of the algorithm. The performances are the ones expected when we wrote the program. Quite surprisingly, we got several times super linear speedup, despite the fact our sequential algorithm is optimal (or almost).

An easy explanation of this phenomenon, is to notice that the size of the search spaces on several processors are in the inverted proportion of their number. As a consequence, each processor uses less memory, requires less garbage

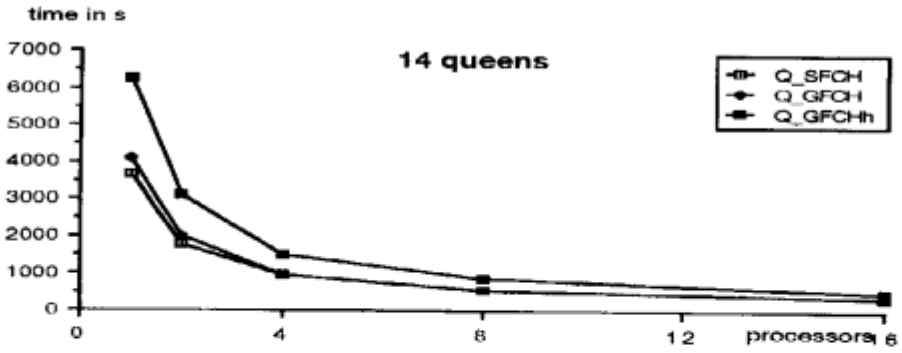

Figure 2.12: 14 queens (365596 solutions)

program		*15 queens*	*16 queens*
	solutions	2279184	14772512
Q_SFCH	time [s]	2215	13727
	time [hms]	36' 55"	3 h 48' 47"
	Mred	841	5526
Q_GFCH	time [s]	1908	11842
	time [hms]	31' 48 "	3 h 17' 22"
	Mred	495	3122

Table 2.21: Queen Program: 15 and 16 queens with 16 processors

collections, and the cache memory of the processor turns out to be more efficient. If the load balance of the processors is ideal, small super-linear speedups may be explained in this manner.

Super-linear performances of the *Q_GFCH*, making the 14 queens problems in 4.21 times faster than with one processor, needs additional explanations. We noticed the decrease of the number of reductions performed by the program, when the number of processors increased (cf. Table 2.20). Our principle of parallelism introduced more constraints on each of the processors, thus the generalized forward checking with first fail principle we used, will detect sooner the impossibilities in the parallel world than in the sequential one, so an increasing number of reductions are saved when the number of processors increases. Notice that this phenomenon appears with 14 queens and more. It seems that reductions to perform the duplication of all tables on each of the processors are not to be under estimated before 13 queens. With 14 and more queens this work may be negligible, compared to the forward checking itself.

program	number of processors				
	1	2	4	8	16
10 Queens time [ms]	6.7	3.6	1.9	1.34	1.18
12 Queens time [ms]	8.1	4.0	2.1	1.32	0.97
13 Queens time [ms]	8.4	4.5	2.5	1.45	0.88
14 Queens time [ms]	10.0	4.9	2.6	1.40	0.94*
15 Queens time [ms]					0.84*
16 Queens time [ms]					0.80*

Table 2.22: Average time for 1 solution with the best of Q_SFCH and Q_GFCH

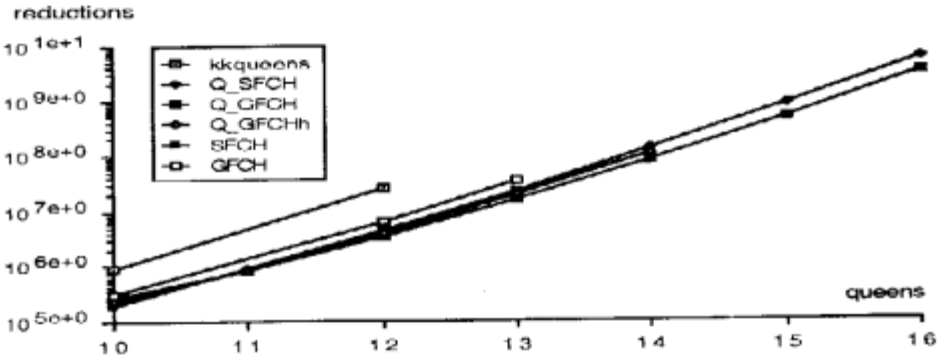

Figure 2.13: Number of reductions

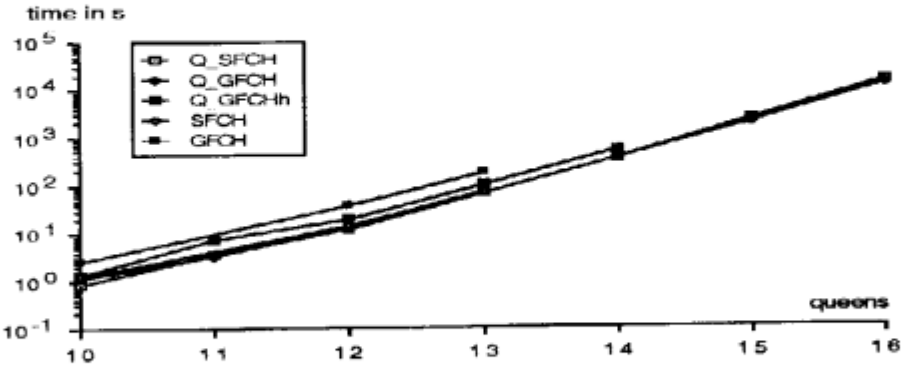

Figure 2.14: time performance with 16 processors

Chapter 3

Conclusions

Four different problem solver have been described and implemented, namely SFCH1, GFCH1, SFCH, GFCH. The Two first ones perform unique solution search in the frame of *speculative AND-parallelism*, the last ones make exhaustive search in the frame of *OR-parallelism*.

The methodology we introduced for the load balancing is very simple, it is to split the search space on several processors by giving a different initialization to each of them. This initialization is expressed under form of *static constraints*. This is a static load balancing.

In the following, we make a short resume of the main characteristics of the algorithms, we give some hints and criticize them so to introduce the future developments to be done. We will see in turn the expression of the constraints, compare the principles of SFCH and GFCH, look at the types of parallelism, and eventually give the performances.

3.1 The Constraint Language

The *Problem Solvers* we presented may be used for any *finite domain CSP* problem. The problem dependent constraints are generated outside of the *Problem Solver*. The latter generates those constraints, compiles them into efficient bit-encoding and starts the searches.

In aiming to use one of the *Problem Solver*, one needs only to write the constraint generation, which is of course problem dependent. Our constraint language, enters the constraint instances, under a rather naive form. The constraint generation program may be written in a naive manner for both reasons, the constraint generation is often a negligible task compared to the search task, these constraints are compiled into bit-encoding by the *Problem Solvers*.

3.2 Simple versus Generalized Forward Checking

Through the whole study we made some competition between the SFCH and GFCH. Most of the papers we were reading in the field of CSP claim the superiority of the GFCH over SFCH. On one hand, GFCH shows a nice and clever principle, on the other hand SFCH exhibits a great simplicity and permits efficient implementations. Thus, comparing SFCH and GFCH, is an illustration of the dilemma between implementation and theory. As this question is beyond the scope of our study, we show some results of the algorithms.

It is difficult to predict the relative performances of SFCH and GFCH, a case by case study is required. The problem solver are homogeneous in their interfaces, thus we report the choice of the problem solver to the user, and suggest an experimental approach.

3.2.1 Unique solution search

SFCH1 and GFCH1 perform unique solution search. The parallelism they use is *speculative AND-parallelism*. The data are spread to a list of processors, each of them generates locally the constraints, and begins the search in a subspace of the search space. As soon as one of them finds a solution, it sends a stop signal to all other processors, and its result is returned.

3.2.2 Exhaustive solution search

SFCH and GFCH perform exhaustive solution search. The parallelism they use is *OR-parallelism*. The data are spread to a list of processors, each of them generates locally the constraints, and begins the search in a subspace of the search space. The results of the processors are merged to give the solution of the algorithm.

3.3 Parallel Forward Checking

The philosophy of parallelism we used, is to reduce as much as possible the communications in the machine. Thus we use a very coarse grain parallelism. This has a major advantage, the overhead of data-spreading may be negligible, thus the performances of our parallel program running on one processor are the same as those of the optimal sequential algorithm.

3.3.1 Types of parallelism

We implemented two types of parallelism, *speculative AND-parallelism* and *OR-parallelism*. Even by using several kinds of parallelism, we kept the same load balancing principle. Because we needed to reduce as much as possible the communications in the machine, each of the processor generates locally, and in redundancy, the constraints of the problem. As this is done sequentially, it affects the overall performances of the parallel implementation (Amdahl's law). This choice has however two major reasons, the first is that it eases the task of writing the constraint generation (done for each application), the second is that the constraint generation is in most of the cases negligible before the other tasks.

3.3.2 Load balance

A very simple static load balancing method has been used. It splits the work between the processors by splitting the domain of some variables.

When making a unique solution search using the *speculative AND-parallelism*, this load balancing method is almost ideal, its overhead is small and splits the domain of search into several *a priori* equal sub-domains.

However, the study of the load balancing showed very clearly the limitation of this method, when used to perform the exhaustive solution search. The obtained speedup reach a maximum of 10 with 16 processors. This is the bottleneck of the study; if we want to improve performances we have to focus our attention on load balancing. The correlations between speedup and load balancing have been highlighted by our measures.

3.4 Performances

The performance of the *Parallel Problem Solver* has been demonstrated on four applications for both, unique and exhaustive solution search. Some properties of these algorithms could be observed.

3.4.1 Unique solution search

queens For tiny search spaces (until 15 queens) SFCH1 takes advantage of its simplicity and is between 2 and 3 times faster than GFCH1. Conversely, GFCH1 is much fastest for large search spaces, and may be 1000 times faster than SFCH1.

zebra As the zebra is solved in a very short time (0.48 seconds), both algorithms show similar performances.

maze According to the shape of the maze, we found comparable results between SFCH1 and GFCH1, or several degree of magnitude difference in the advantage of GFCH1.

Sometimes SFCH1 takes advantage of *speculative AND-parallelism*, achieving an excellent *** speedup with 16 processors on the 30 queen problem. But as said by its name, this parallelism does not insure any speedup in the general case. Quite often we got no speedup (but no slowdown), but speedups are not limited by the ideal linear case, as proven by the 30 queens.

GFCH1 gets no, or very low speedup in parallel, its principle leads it anyway efficiently to the solution. Thus speculative search is or minor impact.

3.4.2 Exhaustive solution search

queens SFCH turns out to be the fastest, outperforming GFCH by a factor of 2 or 3. The speedups are up to 10 with 16 processors. The search of the 724 solutions of the 10 queens is performed in 5.54 seconds on one processor, and 1.25 seconds on 16 processors.

zebra The solution of the zebra problem is found in .48 seconds on one processor, the top speed is obtained with 2 processors, in 0.4 seconds. SFCH and GFCH exhibit very similar performances.

scheduling In scheduling problem, SFCH turns out to be more efficient than GFCH, a factor of 2 or 3 was noted in our examples. The speedups obtained in scheduling are very poor, reaching only a factor of 3 with 16 processors.

maze By studying several mazes, we could illustrate the force of SFCH and GFCH. In some cases SFCH is 3 times faster than GFCH, whereas in other examples GFCH is 1500 times faster than SFCH. The speedups attain 6 in the mazes.

In most of the examples we showed, SFCH outperforms GFCH by a factor of 2 or 3. In some tricky cases GFCH is winning by several orders of magnitude. SFCH is fast because very simple, whereas GFCH gains efficiency on large search spaces.

This trade-off between both algorithms partly due to the implementation language, as we could show it during the comparison to related work.

3.4.3 Related work

We compared the performances of our *Problem Solver* SFCH and GFCH to specialized programs solving the queen problem, Q_SFCH and Q_GFCH, using the same principles. The specialized programs show very closed performances, Q_GFCH outperforms Q_SFCH with more than 14 queens. The SFCH problem solver has performances close to Q_SFCH whereas GFCH is two times slower than Q_GFCH. This slow down is due to some inefficiencies in KL1 (see section 3.5).
When comparing to other studies in *Parallel Logic Programming* our algorithms perform quite well, often faster in raw speed, but with less speedup than other algorithms. Let us emphasize that we wanted to make efficient algorithms, even when running on one processor. Thus it becomes very difficult to achieve as well linear speedups.

3.5 ESP version

The *Parallel Forward Checking Problem Solver* have been translated into ESP by Satoshi Terasaki. The performances in ESP are better than in KL1 on one processor. SFCH and SFCH1 gain 10% in speed whereas GFCH and GFCH1 gain approximately 50%.

Speed improvements are not due to simplification of the program, (the parallelism has been removed), but come rather from the ESP compiler, and ESP's stack mechanism. The performances of SFCH and GFCH are close, even for exhaustive search.

3.6 Future Improvements

Besides interesting ideas of improvements, we can still improve the existing code. The current KL1 compiler seems quite archaic is some ways, and we passed several days to find some way to overcome unexpected inefficiencies.

We dare think that using code generation, could speedup the algorithms, especially GFCH and GFCH1 by a factor of 2. In fact, if this programs would be in LISP code, I would use the macro facilities to generate the most efficient code performing the `bit_sum` and `first_bit` operations. Thus we would reach the same performances as the specific queen program, even with the *Problem Solver*.

3.6.1 Unique solution search

The SFCH1 unique solution search algorithm, may be improved by introducing some *Monte-carlo* methods. Since there is no accurate order to propagate the constraints of the variables, it is certainly interesting to implement a random choice of this order; so that each of the processor will propagate the constraints in its own manner. Thus we

increase the chances to find an accurate constraint propagation by using the frame of the *speculative AND-parallelism*.

This improvement is very easy to do, but requires extensive experimental studies so to obtain statistically sound results.

3.6.2 Exhaustive solution search

During the whole study we could notice the limitation coming from the static load balance we used. It is important to improve this load balance; but we have to be careful in doing so. Let us remind the high price of communication in the Multi-PSI. As we want to avoid big overheads due to communications, we will conceive a kind of *Not too bad load balance*, carrying an approximate work out, in the coarse granularity level we used during this study.
One could see some similarities between this new load balance and the method called *Kabu-Wake* developed at Fujitsu Laboratories[6, 7]. It is to make an evaluation of the work to be performed on the processors, and then spread it. In the frame of our work, it seems important to apply this method in a coarse grain environment, as well in the dimension of the data as the time dimension. Our policy of programming the Multi-PSI remains unchanged: minimizing the communications in the parallel world.

Bibliography

[1] B. Burg. Inductive learning in a parallel environment. In *France-Japan Artificial Intelligence and Computer Science Symposium*, 1989.

[2] B. Burg and D. Dure. FLIB user manual. Technical Report TR 529, ICOT, 1990.

[3] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the parallel inference machine operating system (pimos). In *Proceeding of the International Conference on Fifth Generation computer Systems*, 1988.

[4] P. Van Hentenryck. Parallel constraint satisfaction in logic programming. In *Sixth International Conference on Logic Programming, Lisbon*, 1989.

[5] P. Van Hentenryck and M. Dincbas. Forward checking in logic programming. In *Fourth International Conference on Logic Programming, Melbourne*, 1987.

[6] K. Kumon, H. Masuzawa, A. Itashiki, K. Satoh, and Y. Sohma. KABU-WAKE: A new parallel inference method and its evaluation. In *COMPCON 86 Spring*, 1986.

[7] H. Masuzawa, K. Kumon, A. Itashiki, K. Satoh, and Y. Sohma. KABU-WAKE: parallel inference mechanism and its evaluation. In *Fall Joint Computer Conference*, 1986.

[8] K. Nakajima, Y. Inamura, N. Ichiyoshi, K. Rokusawa, and T. Chikayama. Distributed implementation of KL1 on the Multi-PSI/V2. In *Proceedings of the Sixth International Conference on Logic Programming*, 1989.

[9] P. G. Paulin and J.P. Knight. Force-directed scheduling in automatic data path scheduling. In *24th DAC*, 1987.

[10] P.G. Paulin, J.P. Knight, and E.F. Girczyc. Hal: A multi-paradigm approach to automatic data path synthesis. In *23rd DAC*, 1986.

[11] S. Samal and T. Henderson. Parallel consistend labeling algorithms. *International Journal of Parallel Programming*, 16(5), 1987.

[12] R. Seidel. A new method for solving constraint satisfaction problems. In *the seventh IJCAI*, 1981.

[13] T. Sugimoto. Csp on the multi-psi. Icot, 90.

[14] E. Tick. Performance of parallel logic programming architectures. Technical Report TR 421, ICOT, 1988.

[15] K. Ueda. Introduction to guarded horn clauses. Technical Report TR 209, ICOT, 1986.