TR-595

Parallel Forward Checking
Second 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

Second part

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

Bernard Burg

August, 1990

Abstract

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.

Their principle, sequential implementation, parallel implementation, user guide have been described in the first part of this report. Application results have as well been demonstrated on commonly known problems: the queens, the zebra (also called the 5 houses problem), scheduling and mazes. They have been run on the Multi-PSI, full results have been reported.

The second part of the report describes the experimentation we made by introducing *dynamic load balancing*. Its principle is described, the choice of the communications between processors and their syntax are given. The following chapter describes in detail the constraints generation of the examples shown in the first report. The code of the constraint generation is reported. Then we publish the code of the problem solvers. They are written in KL1[4], run under PIMOS[2] on the Multi-PSI[3] and perform the *Parallel Forward Checking*.

Contents

List of Figures

Chapter 1

Dynamic Load Balancing

The first part of the report described the applications of *Parallel Forward Checking*. When used to make the exhaustive solution search, we noticed sometimes poor speedups. We could show a strong correlation between the speedups and the load balancing of these applications. Thus, to improve the efficiency of the parallel version of *Forward Checking*, we have to improve load balancing. One of the ways of doing so would have been to improve the static load balance, however this would lead to a case by case study without generality, as a consequence we undertook the more difficult study of dynamic load balancing.

1.1 Principle

To achieve efficiency in the parallel world on the Multi-PSI[3], we adopted so far the principle of minimizing the communications in the machine since inter-processor communications are slow. The parallel versions of *Forward Checking* show speedup to the optimal sequential version because the communications have been reduced to their strict minimum, by using static load balance. Implementation of dynamic load balance has to make some compromises between the costs of communications and accuracy of the load balance. In the following two sections we discuss our choices for the granularity and the type of communications between processors.

1.1.1 Granularity: coarse grain

Loosely coupled machines, like the Multi-PSI are devoted to deal with coarse granularity problems. The main problem of the Multi-PSI comes from its slow data-transmission between processors. As each of the processors has its own memory, transmission is to transform the local atom-numbers into atoms, transmit them and encode them into the new local atom numbers. This operation is performed by the firmware and is in consequence slow. If programming efficient algorithms, one has to avoid as much as possible the communications.

As a consequence we will not change the granularity of the static implementation, but add a dynamic load balance so to share roughly the tasks.

1.1.2 Communications between processors

Coarse grain parallelism limits the frequency of the communications, but requires the circulation of several types of messages. To do so, there are two wildly used techniques in competition.

Token ring

This principle is to put a token in a ring, going from one processor to the others. A processor communicates only with the token, and has to wait it to exchange information with other processors. This principle is simple and makes an easy control of the coherence of the messages, via control of the token. If there is no message to transmit, the token goes empty from processor to processor, thus it may slowdown the algorithm by making unnecessary communications.

Asynchronous communications

Each of the processor may emit a message towards his neighbors, and as well receive messages from them. All communications are asynchronous, the control of the coherence of the whole state of the machine becomes difficult.

1.1.3 Outline of the algorithm

The *Parallel Forward Checking* problem solver begin their work by using the static load balancing defined in the preceding part of the report. All processors begin their work until one of them finishes it. Then this processor emits an **ask** signal, going to all the processors, asking them to make an estimation of the remaining work to be done. When this signal comes back to the asking processors, the latter has an image of the overall tasks to be done. Locally, this processor chooses the most busy of the processors, and sends it a **split** signal, so to share its work. When this processor gets the **split** message, it makes again an estimation of the remaining work to be performed, and then splits it into two equal parts. One of this parts is send to the asking processor, the other is done by the locally. The Figure 1.1 gives a simplified synoptic of the principle used for the dynamic load balancing. The principle as described

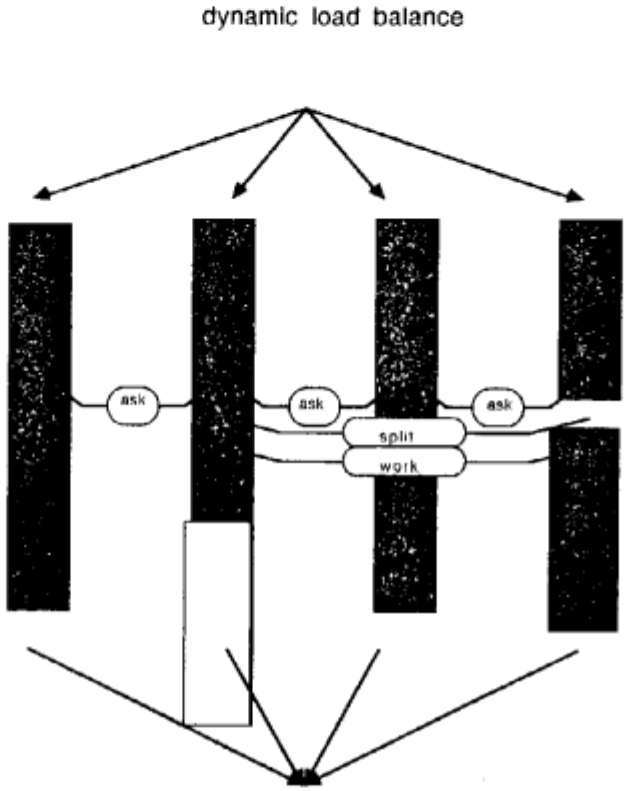

Figure 1.1: Outline of the Dynamic load balance

would lead to an increase of communications as the algorithms reaches the end of its work because the size of the tasks transmitted between processors decrease in a geometric sequence, always half of the work is transmitted. Thus we introduced an additional parameter in our algorithms, giving a threshold after which the dynamic load balance switched to static load balance. This message is the **static** message. Once a processor reached this critical point, it sends this message to all its fellows so to finish the work without wasting time in communicating negligible tasks.

1.2 Implementation

Implementation of the preceding outline poses some technical problems. The first is to know how to make the estimations of the remaining work on the processors. Then the communications between processors have to be created so to minimize them, and transmit all the different messages. The communications modifies in turn the algorithms, so they may be able to suspend their work, treat the communications and continue the work. We describe these points in the following sections.

1.2.1 Estimation of work

One estimation of the work to be done by the algorithm, is to consider the the size of the search space $D_T = D_1 \times \ldots \times D_n$. At the begin of the algorithm, the static load balancing splits this search space into Pnb (number of processors) equal subspaces. The static load balance is optimal if the sparseness of the solutions in these subspaces,

is equal.

As the *Forward Checking* progresses in its task, it keeps in memory the current point of the searches. It is quite simple to take this current point in the search space and to calculate its "distance" to the end of the search.

This estimation from the current point to the end of the search space is a measure of the remaining work of the processors. In fact, as these measures are performed almost simultaneously on the processors and each of the processors being equivalent, our measure gives an image of the sparseness of each of the domains. As a consequence, dynamic load balance will improve the performance of the problems showing poorly balanced sparsenesses between the solution subspaces of the processors.

1.2.2 The communications

We chose to implement the asynchronous communications frame in the Multi-PSI. The processors are related by two kinds of streams. On of them merges the results calculated by the processors, the other stream connects all the processor in a ring and assures the communications between them. Next we study the messages circulating in this stream and explain their interaction with the work of the processors.

ask

The $[ask, Pro_a]$ message is emitted by the idling processor Pro_a. The next processor in the ring gets this message and makes the estimation of the remaining work. It adds it to the message before forwarding it to the following processor, the message is then $[ask, Pro_a, \{Pro_x, Estim_x\} \ldots]$.
When this message comes back to the Pro_a processor, the latter has all the information on the state of advance of the work, and chooses the most busy processor so to split its task by issuing the `split` signal.

split

The syntax of the this message is $[split, Pro_s, Pro_a]$, which means to split the work of the Pro_s and send half of the remaining work to the Pro_a processor. As long as this message is read by another processor than Pro_s, it is simply duplicated. When it reaches the Pro_s processor, then the latter makes updates the estimation of the remaining work and splits it into two equal parts. The first of them is done locally, the second part is transmitted to the Pro_a processor, via the **work** signal.

work

The work signal $[work, Pro_a, Begin, End]$ transmits new work to the idling processor Pro_a, with the lower and higher limits of the search space. As for the **split** signal, this message is copied by the processors, until it reaches its destination, the Pro_a processor. This one begins then its work.

static

The static message may be issued in several conditions, each time the remaining work to be done is inferior to the given threshold Lb, when the asking processor Pro_a performs the load balance, and when the splitting processor Pro_s reached the end of its work. The syntax of this message is $[static, Pro_x \ldots]$, so to detect when this message visited all the processors.

The static message switches the visited processors into **static** *Forward Checking*, cutting them form the communication stream, provided these processors have not send a *split* message. The processors having send a split message stay in the dynamic mode, so to perform the part of work which will be send to them by the **work** signal.

Chapter 2

Examples of Constraints generation

We show in this chapter how we generate the constraints for the examples. We begin with the simplest example, the queens, and go into more complicated ones, the mazes and finaly the scheduling. The principle is first described, then the code is commented.

2.1 Queens

As this is the first example, we give as well the constraints generation as the code calling the problem solvers.

2.1.1 Call of the Problem solvers

```
:- module queens.
:- public constraints/2, sfch/3, gfch/4, sfch1/3, gfch1/4.

% the problem start:
% N is the number of queens
% PNb is the number of processors
% File is the name of the result file. It is a string.
sfch(N, PNb, File):- true|
      sfch:run({N, N}, 'queens', 'number', PNb, File).

gfch(N, WZ, PNb, File):- true|
      gfch:run({N, N, WZ}, 'queens', 'number', PNb, File).

% the same, but with 1 solution search in
% speculative  AND-parallelism

sfch1(N, PNb, File):- true|
      sfch1:run({N, N}, 'queens', 'number', PNb, File).

gfch1(N, WZ, PNb, File):- true|
      gfch1:run({N, N, WZ}, 'queens', 'number', PNb, File).
```

2.1.2 Description of the constraints

variables

The variables are the lines x_i of the board.

domains

The domains of the variables are the contents of the lines, their instances x_j.

static constraints

no static constraints for the queen problem.

dynamic constraints

For an instance of a queen, in line x_i and column x_j, the dynamic constraints express all the positions in the board which are attacked by this queen; namely the line x_i, the column x_j and the diagonals passing by the x_i x_j point, as shown in Figure 2.1.

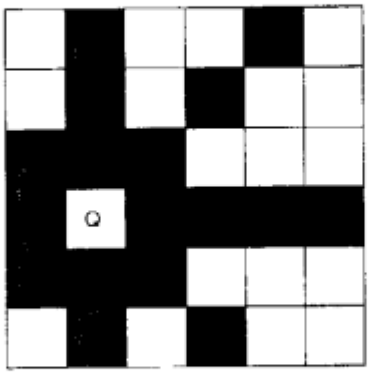

Figure 2.1: The dynamic constraints for queens

2.1.3 Generation of the constraints

```
%
% Constraints of the problem
%
constraints({Size, Size}, Table):- true|
    constraints(Size, 0, [], Table).

constraints(Size, Size, TabIn, TabOut):- true|TabIn=TabOut.
otherwise.
constraints(Size, Col, TabIn, TabOut) :- true|
    init_cols(Size, Size, TabIn, TabIn1, Col),
    constraints(Size, ~(Col+1), TabIn1, TabOut).

init_cols(0, _, TableIn, TableOut, _) :- true|
    TableOut=TableIn.
otherwise.
init_cols(Col, Size, TableIn, TableOut, X) :- true|
    Col1 := Col - 1,
    init_offs(Size, Size, Col1, Elemtab, X),
    TableIn1=[[X, Col1|Elemtab]|TableIn],
    init_cols(Col1, Size, TableIn1, TableOut, X).

    init_offs(0, _, _, Tab, _) :- true|Tab=[].
otherwise.
    init_offs(Off, Size, Col, Tab, X) :-
    Off1 := Off - 1,
    Index := X + Off1,
    Index1 := Index mod Size|
    (Index1=:= Index ->
        const_dom(Index1, Size, Col, Off1, Tab, Tab1);
     otherwise;
        true ->
        const_dom(Index1, Size, Col, ~(X - Index + Size), Tab, Tab1)),
    init_offs(Off1, Size, Col, Tab1, X).
```

```
%
% const_dom(Size, Column, Offset, Pattern)
%
    const_dom(_, _, _, 0, Tab, Tab1) :- true|
    Tab=Tab1.
otherwise.
    const_dom(Index, Size, Col, Off, Tab, Tab1) :-
    Sum := Col+Off,
    Dif := Col-Off|
    const_dom1(Size, Sum, Pat1),
    const_dom1(Size, Dif, Pat2),
    list:append(Pat2, [Col|Pat1], Pat),
    Tab=[[Index|Pat]|Tab1].

    const_dom1(Size, Pos, Pat) :- 0 =< Pos, Pos < Size |
    Pat = [Pos].
otherwise.
    const_dom1(_, _, Pat) :- true| Pat = [].
```

2.2 Mazes

We defined some unusual rules for the mazes. The mazes are entered at the left side, and exit at the right one. There is just one position allowed by column of this maze. When changing columns to progress in the maze, we may only move to the 3 or 5 neighbours in the next column; according to the maze. Some positions are forbidden in the maze, by the walls delimiting the maze.

2.2.1 Description of the constraints

variables

The variables are the columns x_i of the maze.

domains

The domains of the variables are the contents of the columns, their instances x_j.

static constraints

The static constraints define the walls, of the maze, this is to say they restrict the possible moves.

dynamic constraints

For an instance of a position in the maze, in columns x_i and line x_j, the dynamic constraints express all the positions in the maze which cannot be attained from there. Thus, the dynamic constraints remove the domains as shown in Figure 2.2.

2.2.2 Generation of the constraints

```
%
% Constraints of the problem
%
constraints({SizeX, SizeY}, Table):- true|
    static_constraints(SizeX, SizeY, [], ST_cons),
    dynamic_constraints(SizeX, 0, SizeY, ST_cons, Table).

static_constraints(SizeX, SizeY, In, Out):-
  SizeX4 := SizeX/4|
    horizont([-(SizeY-3), -(SizeY-4)], SizeX4, SizeX, In, In1),
    vertic(-(SizeY-5), SizeY, -(SizeX4-2), [], In1, In2),
```

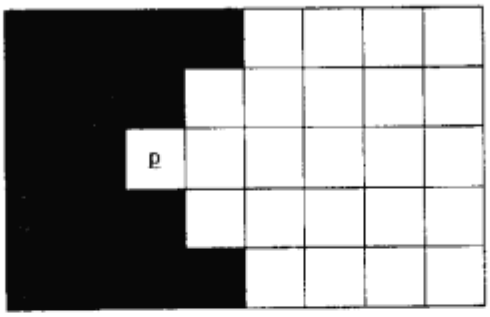

Figure 2.2: The dynamic constraints for maze

```
vertic(0, ~(SizeY-2), ~(SizeX-1),[], In2, Out).

% removes an horizontal line (double, with several elements in Y) from the
% domain. The line is removed form Xbegin to Xend.
%
horizont(Y, Xbegin, Xend, In, Out):- Xbegin < Xend|
    In1=[[Xbegin, 0, [Xbegin | Y]]|In],
    horizont(Y, ~(Xbegin+1), Xend, In1, Out).
otherwise.
horizont(_, _, _, In, Out):- true|In=Out.


% vertic removes a vertical line from the domain,
% vertic(Begin, End, Res, In, Out),
% begin is the begin of the wall, End is the end.
% to make a real door in a wall, we have to apply twice this predicate.

vertic(Cur, Max, X, Cons, In, Out):- Cur < Max|
Cons1=[Cur|Cons],
    vertic(~(Cur+1), Max, X, Cons1, In, Out).
vertic(Max, Max, X, Cons, In, Out):- true|
    Out= [[X, 0, [X |Cons]]|In].


dynamic_constraints(SizeX, SizeX, _, TabIn, TabOut):- true|TabIn=TabOut.
otherwise.
dynamic_constraints(SizeX, Col, SizeY, TabIn, TabOut) :- true|
    init_cols(SizeY, SizeX, SizeY, TabIn, TabIn1, Col),
    dynamic_constraints(SizeX, ~(Col+1), SizeY, TabIn1, TabOut).

init_cols(0, _, _, TableIn, TableOut, _) :- true|
    TableOut=TableIn.
otherwise.
init_cols(Col, SizeX, SizeY, TableIn, TableOut, X) :- true|
    Col1 := Col - 1,
    init_offs(SizeX, SizeX, SizeY, Col1, Elemtab, X, Sync),
    TableIn1=[[X, Col1|Elemtab]|TableIn],
    (wait(Sync) ->
      init_cols(Col1, SizeX, SizeY, TableIn1, TableOut, X)).

    init_offs(0, _, _, _, Tab, _, Sync) :- true|Tab=[], Sync=1.
```

```
otherwise.
    init_offs(Off, SizeX, SizeY, Col, Tab, X, Sync) :-
    Off1 := Off - 1,
    Index := X + Off1,
    Index1 := Index mod SizeX|
    (Index1=:= Index ->
        const_dom(Index1, SizeY, Col, Off1, Tab, Tab1, LSync);
     otherwise;
        true ->
        const_dom(Index1, SizeY, Col, ~(X - Index + SizeX), Tab, Tab1, LSync)),
    init_offs1(Off1, SizeX, SizeY, Col, Tab1, X, LSync, Sync).

init_offs1(Off1, SizeX, SizeY, Col, Tab1, X, LSync, Sync):- wait(LSync)|
    init_offs(Off1, SizeX, SizeY, Col, Tab1, X, Sync).

%
% const_dom(SizeX, Column, Offset, Pattern)
%
    const_dom(_, _, _, 0, Tab, Tab1, Sync) :- true|
    Tab=Tab1, Sync=1.
otherwise.
    const_dom(Index, SizeY, Col, OffIn, Tab, Tab1, Sync) :-
    Off := OffIn * 2,
    Sum := Col+Off+1,
    Dif := Col-Off|
    (Dif < 1, Sum >= SizeY -> Tab=Tab1, Sync=1;
     otherwise;
    true ->
        generate(0, Dif, [], Pat1, _),
        generate(Sum, SizeY, Pat1, Pat, Sync),
        Tab=[[Index|Pat]|Tab1]).

generate(Min, Max, PatIn, PatOut, Sync):- Min < Max|
    PatIn1=[Min|PatIn],
    generate(~(Min+1), Max, PatIn1, PatOut, Sync).
otherwise.
generate(_, _, PatIn, PatOut, Sync):- true| PatOut=PatIn, Sync=1.
```

2.3 Scheduling

2.3.1 Description of the constraints

variables

The variables are the events x_i to be scheduled.

domains

The domains of the events represent the temporal dimension x_j of the problem. The problem is given in terms of time-steps.

static constraints

Most of the events may only occur inside of a static time-window; according to the *As Soon As Possible ASAP* and *As Late As Possible ALAP*. Lets show the example described in the first part of the report, Figure 2.4. To simplifie, we represent only one part of the problem, generalization is straightforward.

If we ask the system to make the scheduling of this problem in 6 time steps, it begins to determin the *As Soon As Possible* and *As Late As Possible* schedules, as shown in Figure 2.3. The domain of the variables is thus reduced, its limits becomes the ones defined by ASAP and ALAP, $x_1 \in [1,3]$,$x_2 \in [1,3]$, $x_3 \in [2,4]$,$x_4 \in [3,5]$,$x_5 \in [1,4]$,$x_6 \in [2,5]$,

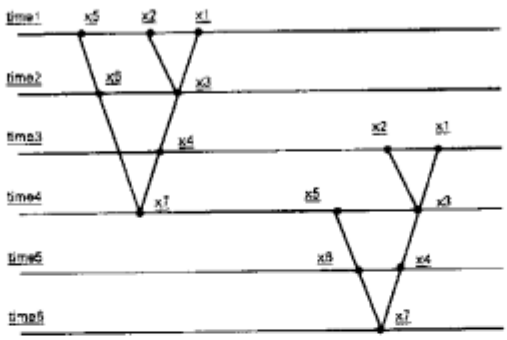

Figure 2.3: The ASAP and ALAP schedulings

$x_7 \in [4,6]$.
The size of the search space is thus dramatically reduced and passes from $6^7 = 279936$ to (respectively) $3^4 \times 4^2 \times 3 = 3888$. This reduction of the search space by a factor of 72 illustrates the usefulness of static constraints.

dynamic constraints

For an instance of an event x_i in the temporal space x_j, the dynamic constraints express all the time-steps of the related events which are invalidated by the current instance; this is to say all the predecessors of this even should be finished before, and all the successors should begin after. This is quite simple to program with the tree structure we adopted to represent the relations between events. It is to search for parent children relations in a graph (classical problem).

2.3.2 expression of the constraints

To be able to express different problems, we made a little language. To express a problem, one has just to express the successor relation, inside of a vector. The positions of the vector are the events, their content gives their successors; sometimes a list if there are several successors. The 0 position of this vector is a virtual element pointing towards all the beginning elements. The end elements point as well to a virtual element being the last position of the vector. The latter always contains an empty list.

```
 for the problem described in Figure 2.4, First part of the report.
       Data={[1,2,5,8,10],3,3,4,7,6,7,12,9,12,11,12,[]}.


 for the problem described in Figure 2.5, First part of the report.
Data={[1,7],[2,3],[4,6],[9,11],5,[11,12,13],[9,12,13],8,[5,9],
             [10,12],[13,17],[16,18],[14,17],16,[15,16],16,19,[14,15],15,[]}.
```

2.3.3 Generation of the constraints

As we already commented the principles, and the programs are quite simple, basically performing parent-children searches, we do not add comments to the sources.

```
constraints({X, Y}, Res):- true|
% these +1 appear because time frame is in the range [1..n],
% and gfch in [0..~(n-1)]
    time_frame(~(X+1), ~(Y+1), Dom, Ch_tree, Pa_tree),
    make_mask(0, Y, Mask),
    static_constraints(Dom, Dom1, 1, ~(X+1), Mask, Mask1,Sta_res),
    transform(Ch_tree, Ch_tree1),
    transform(Pa_tree, Pa_tree1),
% up here, everything is clean again
```

```
    dynamic_constraints(Dom1, Ch_tree1, Pa_tree1, Mask1, Sta_res, Res).

make_mask(Cur, Max, Mask):- Cur < Max|
    Mask=[Cur |Mask1],
    make_mask(~(Cur+1), Max, Mask1).
make_mask(Max, Max, Mask):- true| Mask=[].

static_constraints(Dom, Domc, Cur, Max, Mask, Maskc, Res):-
  Cur < Max,
  Cur1:= Cur-1|
    set_vector_element(Dom, Cur, Elem, _, Dom0),
    set_vector_element(Dom0, Cur1, _, Elem1, Dom1),
    transf_simple(Elem, Elem1, Elem2),
    diff(Elem2, Mask, Mask1, LoRes),
    Res=[[Cur1, 0, [Cur1 |LoRes]]|Res1],
    static_constraints(Dom1, Domc, ~(Cur+1), Max, Mask1, Maskc, Res1).
static_constraints(Dom, Domc, Max, Max, Mask, Maskc, Res):- true|
    Dom=Domc, Mask=Maskc, Res=[].

% transf_simple aand transform are to bring the variables,
% from [1..N], to [0..~(N-1)], as well as the domains.
transf_simple([A|B], R1, R2):-
  A1:=A-1|
    R1=[A1 |R11],
    R2=[A1 |R22],
    transf_simple(B, R11, R22).
transf_simple([], R1, R2):- true|
    R1=[], R2=[].

transform([A|B], Res):- list(A)|
    transform(A, Res1),
    transform(B, Res2),
    Res=[Res1 | Res2].
otherwise.
transform([A|B], Res):- true|
    transform(B, Res1),
    Res=[~(A-1)|Res1].
transform([], Res):- true|Res=[].

diff([A|B],[A|C], Copy, Res):- true|
    Copy=[A|Copy1],
    diff(B, C, Copy1, Res).
otherwise.
diff(B, [A|C], Copy, Res):- true|
    Copy=[A|Copy1],
    Res=[A|Res1],
    diff(B, C, Copy1, Res1).
diff(_, [], Copy, Res):- true|
    Copy=[], Res=[].

dynamic_constraints(Dom, Child, Parent, Mask, In, Out):- true|
    relat('before', Child, Dom, Dom1, Mask, Mask1, In, In1),
    relat('after', Parent, Dom1, _, Mask1, _, In1, Out).

relat(Type, [A|B], Dom, Domc, Mask, Maskc, In, Out):- true|
    relat1(Type, A, Dom, Dom1, Mask, Mask1, In, In1),
    relat(Type, B, Dom1, Domc, Mask1, Maskc, In1, Out).
relat(_, [], Dom, Domc, Mask, Maskc, In, Out):- true|
    Dom=Domc, Mask=Maskc, In=Out.
```

```
relat1(Type, [A|B], Dom, Domc, Mask, Maskc, In, Out):- list(A)|
    relat1(Type, A, Dom, Dom1, Mask, Mask1, In, In1),
    relat1(Type, B, Dom1, Domc, Mask1, Maskc, In1, Out).
otherwise.
relat1(Type, [A|B], Dom, Domc, Mask, Maskc, In, Out):-
  vector_element(Dom, A, DomA)|
    relat2(Type, A, DomA, B, Dom, Dom1, Mask, Mask1, In, In1),
    relat1(Type, B, Dom1, Domc, Mask1, Maskc, In1, Out).
relat1(_, [], Dom, Domc, Mask, Maskc, In, Out):- true|
    Dom=Domc, Mask=Maskc, In=Out.

relat2(Type, X, DomX, [Y|Z], Dom, Domc, Mask, Maskc, In, Out):- list(Y)|
    relat2(Type, X, DomX, Y, Dom, Dom1, Mask, Mask1, In, In1),
    relat2(Type, X, DomX, Z, Dom1, Domc, Mask1, Maskc, In1, Out).
otherwise.
relat2(Type, X, DomX, [Y|Z], Dom, Domc, Mask, Maskc, In, Out):-
  vector_element(Dom, Y, DomY)|
    relat3(Type, X, Y, DomX, DomY, Mask, Mask1, In, In1),
    relat2(Type, X, DomX, Z, Dom, Domc, Mask1, Maskc, In1, Out).
relat2(_, _, _, [], Dom, Domc, Mask, Maskc, In, Out):- true|
    Dom=Domc, Mask=Maskc, In=Out.

relat3(Type, X, Y, [A|B], DomY, Mask, Maskc, In, Out):- true|
    relat4(Type, A, DomY, Const),
    (Const=[] -> In1=In;
     otherwise;
     true ->     In1=[[X, A, [Y |Const]]|In]),
    relat3(Type, X, Y, B, DomY, Mask, Maskc, In1, Out).
relat3(_, _, _, [], _, Mask, Maskc, In, Out):- true|
    Mask=Maskc, In=Out.

relat4('before', Lim, DomY, Const):- true|
    before(DomY, Lim, Const, _).
relat4('after', Lim, DomY, Const):- true|
    after(DomY, Lim, Const, 0, _).

before([A | _], C, Before, Ln):- A > C|
      Before=[], Ln=0.
otherwise.
before([A | B], C, Before, Ln):- true|
      Before=[A | Before1],
      Ln:= Ln1+ 1,
      before(B, C, Before1, Ln1).
before([], _, Before, Ln):- true|
      Before=[], Ln=0.

after([A | B], C, After, Aln, Ln):- A < C|
      after(B, C, After, ~(Aln-1), Ln).
otherwise.
after([A | B], C, After, Aln, Ln):- true|
      After=[A | B],
      Ln:=Aln.
after([], _, After,Aln, Ln):- true|
      After=[], Ln=Aln.

% we have the definition of the problem

time_frame(X, Y, Dom, Ch_tree, Pa_tree):-
```

```
        Dln:=X+1|
        schedul:init(X, Dln, Data),
        new_vector(PIn, Dln),
        parent(Data, Child, PIn, Par, 0, Dln),
        set_vector_element(Par, 0, _, [], Parent),
        asap(Child, 0, Dln, Asap, AsapMax, Ch_tree),
        alap(Parent, Y, Dln, Alap, Pa_tree),
        make_domain(Asap, Alap, Dom, Dsize, Dln).

 parent(Data, Datac, In, Out, Pos, Max):- Pos<Max|
       set_vector_element(Data, Pos, Ele, Ele1, Data1),
       parent1(Ele, Ele1, Pos, In, In1),
       parent(Data1, Datac, In1, Out, ~(Pos+1), Max).
 parent(Data, Datac, In, Out, Max, Max):- true|
       Data=Datac, In=Out.
 parent1([A|B], Copy, Pos, In, Out):- true|
       set_vector_element(In, A, Elem, Resel, In1),
       (Elem=:=0 -> Resel=Pos;
        list(Elem) -> Resel=[Pos|Elem];
        otherwise;
        true -> Resel=[Pos,Elem]),
       Copy=[A|Copy1],
       parent1(B, Copy1, Pos, In1, Out).
 parent1([], Copy, _, In, Out):- true|
       Copy=[], In=Out.
 otherwise.
 parent1(A, Copy, Pos, In, Out):- true|
       set_vector_element(In, A, Elem, Resel, Out),
       (Elem=:=0 -> Resel=Pos;
        list(Elem) -> Resel=[Pos|Elem];
        otherwise;
        true -> Resel=[Pos,Elem]),
       Copy=A.

 % ASAP and ALAP are simply vectors.
 % we make a depth first of child or parent and update the
 % ASAP and ALAP by m'putting the min and max inside.
 % then we can do the csp.
 % csp should work as well without this.

 % depth first takes the Child or PArent and constructs the
 % chronology if the operations in lists.
 % the result Lis is the pre-fixed tree representation
 % it filters as well the element :Filt, this allows to construct
 % the and removing the end and begin, unnecssary in the following.

depth_first(Vect, Vc, Begin, Filt, Lis):- true|
      set_vector_element(Vect, Begin, Elem, Elemc, Vect1),
      depth_first1(Vect1, Vc, Elem, Elemc, Filt, Lis1),
      Lis=[Lis1].
depth_first1(Vect, Vc, [A|B], Copy, Filt, Lis):- true|
      set_vector_element(Vect, A, Elem, Elemc, Vect2),
      depth_first1(Vect2, Vect3, Elem, Elemc, Filt, Lis1),
      Copy=[A | Copy1],
      Lis=[[A |Lis1] | Lis2],
      depth_first1(Vect3, Vc, B, Copy1, Filt, Lis2).
depth_first1(Vect, Vect1, [], Copy, _, Lis):- true|
      Vect=Vect1, Copy=[], Lis=[].
otherwise.
```

```
depth_first1(Vect, Vc, Filt, Copy, Filt, Lis):- true|
      set_vector_element(Vect, Filt, Elem, Elemc, Vect2),
      depth_first1(Vect2, Vc, Elem, Elemc, Filt, Lis),
      Copy=Filt.
depth_first1(Vect, Vc, A, Copy, Filt, Lis):- Filt =\= A|
      set_vector_element(Vect, A, Elem, Elemc, Vect2),
      depth_first1(Vect2, Vc, Elem, Elemc, Filt, Lis1),
      Copy=A,
      Lis=[A |Lis1].

% for As Soon As Possible
% we make a vector and scan the list of Children,
% the maxi of the time schedule is set to the ASAP vector

asap(Child, Begin, Ln, Res, Max, Ch_tree2):- true|
      depth_first(Child, _, Begin, Ln, Ch_tree),
      new_vector(ResIn, Ln),
      asap1(Ch_tree, Ch_tree1, ~(Begin+1), ResIn, Res1),
      list:rec_remove(Ch_tree1, ~(Ln-1), _, Ch_tree2),
      set_vector_element(Res1, ~(Ln-1), Max, Max, Res).

asap1([A |B], Copy, Begin, VecIn, VecOut):- true|
      (list(A) -> asap1(A, Cop1, Begin, VecIn, VecIn1),
                  asap1(B, Cop2, Begin, VecIn1, VecOut),
                  Copy=[Cop1 | Cop2];
       otherwise;
       true -> set_vector_element(VecIn, A, Val, NewVal, VecIn1),
               fel:maxi(Val, Begin, NewVal),
               Copy=[A | Cop],
               asap1(B, Cop, ~(Begin+1), VecIn1, VecOut)).
asap1([], Copy, _, VecIn, VecOut):- true|
      VecIn=VecOut, Copy=[].

% for As Late As Possible
% we make a vector and scan the list of Parents
% the mini of the time schedule is set to the ALAP vector
% to do so, this vecor has been instanciated to the Max value of
% the time schedule, issued by the asap predicate.

alap(Parent, Maxval, Ln, Res, Pa_tree1):- true|
      depth_first(Parent, _, ~(Ln-1), 0, Pa_tree),
      new_vector(ResIn1, Ln),
      alap0(ResIn1, ~(Ln-1), Maxval, ResIn),
      alap1(Pa_tree, Pa_tree1, ~(Maxval-1), ResIn, Res).
alap0(Vect, Cur, Max, VectR):- Cur > -1|
      set_vector_element(Vect, Cur, _, Max, Vect1),
      alap0(Vect1, ~(Cur-1), Max, VectR).
alap0(Vect, -1, _, VectR):- true|
      Vect=VectR.

alap1([A |B], Copy, Maxval, VecIn, VecOut):- true|
      (list(A) -> alap1(A, Cop1, Maxval, VecIn, VecIn1),
                  alap1(B, Cop2, Maxval, VecIn1, VecOut),
                  Copy=[Cop1 | Cop2];
       otherwise;
       true -> set_vector_element(VecIn, A, Val, NewVal, VecIn1),
               fel:mini(Val, Maxval, NewVal),
               Copy=[A | Cop],
               alap1(B, Cop, ~(Maxval-1), VecIn1, VecOut)).
```

```
alap1([], Copy, _, VecIn, VecOut):- true|
      VecIn=VecOut, Copy=[].

% make_domain prepares the domain for the Csp.
% it outputs the variation domain of each variable, in the Domain vector,
% as {0, [2,3],[1]....} and the list Dsize containing the length of each
% domain and the variable name. [[2,1],[1,2]...]
% notice that we lose the first and last element of ASAP and ALAP,
% since the begin and end are fixed elements
make_domain(Asap, Alap, Dom, Dsize, Dln):- true|
      new_vector(Dom1, ~(Dln-1)),
      make_domain1(Asap, Alap, Dom1, Dom, Dsize, ~(Dln-2)).
make_domain1(Asap, Alap, DomIn, DomOut, Dsize, Cur):-
  Cur > 0,
  vector_element(Asap, Cur, Sel),
  vector_element(Alap, Cur, Lel)|
      gene_list(Sel, Lel, List, Lln),
      set_vector_element(DomIn, Cur, _, List, DomIn1),
      Dsize=[[Lln, Cur]|Dsize1],
      make_domain1(Asap, Alap, DomIn1, DomOut, Dsize1, ~(Cur-1)).
make_domain1( _, _, DomIn, DomOut, Dsize, 0):- true|
      DomIn=DomOut, Dsize=[].

gene_list(A, B, Lis, Ln):- A < B|
      Lis=[A | Lis1],
      Ln:=Ln1+1,
      gene_list(~(A+1), B, Lis1, Ln1).
gene_list(A, A, Lis, Ln):- true|
      Lis=[A], Ln:=1.

critical_time(X1, Y1, NewY):-
% here again we have these +1 operations
% for historical reasons, the scheduling works between 1 and N
% the problem solver between 0 and (N-1)
      X:= X1+1,
      Y:= Y1+1,
      Dln:=X+1|
      schedul:init(X, Dln, Data),
      new_vector(PIn, Dln),
      parent(Data, Child, PIn, _, 0, Dln),
      asap(Child, 0, Dln, _, Critical, _),
      (Critical > Y -> NewY := Critical - 1,
                       Blabla=['Critical time', NewY],
                       util:p_console(Blabla, _);
       otherwise;
       true ->         NewY := Y1).
```

Chapter 3

Source code of Parallel Forward Checking

The source code is divided into 5 modules, **sfch1, gfch1, sfch, gfch, fch**. The four first ones contain the sources of the problem-solvers, the last module has some utilities used by the problem solvers.

As the code is clear enough, no comment has been added besides the comments already contained in the source files.

The the code uses the facilities provided by the FLIB library[1]. We make an extensive use of bit-encoding, so to achive efficiency.

3.1 SFCH1

```
% Program:    SIMPLE FORWARD CHECKING IN PARALLEL
%
%                unique solution search
%
%                SPECULATIVE AND PARALLELISM
%
%   Author:     Takashi Chikayama (ICOT)
%   Date:       15 Aug 1988
%   Note:       Queens Problem Solver
%               Originally, this was a sequential implementation of the
%               queen prolem
%
%   modified    Bernard Burg
%               Extention to parallel processing
%   Date:       10 May 90.

%   modified    Bernard Burg
%               Extension to simple forward checking (application indpt)
%   Date:       13 June 90.

%   modified    Bernard Burg
%               One solution search with AND parallelism
%   Date:       29 June 90.

%
%   For keeping candidate positions, a bit table is used.
%   Thus, only up to 32 variables problem can be solved.
%


:- module sfch1.
```

```
:- public main/5, run/5, psfch1/5.

% measures for the multi-psi
run({VarNbIn, DomSZ}, PrName, OutF, ProNbIn, File):- true|
      fch:fch_input(VarNbIn, VarNb, ProNbIn, ProNb),
      Name=string#"SFCH_ONE",
      fel:get_code(sfch1, psfch1, 5, Qcode, normal),
      util:start_stats(Qcode, {{VarNb, DomSZ}, PrName, OutF, ProNb, L}, STR),
      STR1=[{0, string#"problem solver", Name},
            {0, string#"problem", PrName},
            {0, string#"size", {VarNb, DomSZ}},
            {0, string#"processors", ProNb},
            {0, string#"solutions", L}|STR],
     (string(File, _, _) ->  fch:write_in_file(STR1, File);
      alternatively;
      true -> File=L).

psfch1(N, PrName, OutF, ProNb, Res):- true|
      par:create_list_of_p(0, ~(ProNb-1), {N, PrName, OutF, ProNb}, Data),
      par:apply_list_of_p(Data, _, sfch1, main,
         {data_in, processor_in, merge_out, stream_in, stream_out},
          Res1, Circ, Circ),
      list:sync_sum(Res1, Res2, Tot),
      (wait(Tot) ->
          util:req_time(string#"time", Sync),
          psfch11(OutF, Tot, Res2, Res, Sync)).

psfch11(OutF, Tot, Res2, Res, Sync):- wait(Sync)|
          util:req_red(string#"reductions", Sync1),
          (wait(Sync1), OutF=all -> Res=Res2;
           otherwise;
           true ->      Res=[Tot, Res2]).

main({Size, PrName, OutF, MaxPro}, CurPro, ME_out, ST_in, ST_out) :- true|
      main1({Size, PrName, MaxPro}, CurPro, Ans, ST_in, ST_out),
      list:sync_length(Ans, AnsC, LLn),
      (wait(LLn), OutF=all -> ME_out=AnsC;
       otherwise;
       wait(LLn)->   ME_out=[LLn]).


    main1({{SizeX, SizeY}, PrName, MaxPro}, CurPro, Ans, ST_in, ST_out) :-
    init_nondets(SizeX, SizeY, MaxPro, CurPro, Nondets, PT),
    fch:single_bit_table(SBT),
    fel:get_code(PrName, constraints, 2, PRCode, normal),
    fel:shoen_execute(PRCode, {{SizeX, SizeY}, Cons}, 0, 4096, 1,
    [start|Ctl], Rpt),
    fch:control_shoen(Rpt, Ctl, SSync),
    fch:mpattern_table(Cons, SizeX, SizeY, PT, SSync, CurPro),
    new_vector(AnsV, SizeX),
    choice(Nondets, AnsV, [], Ans, SBT, PT, ST_in, ST_out).

    choice(_, _, Sin, S, _, _, ST_in, ST_out):- ST_in=[stop|_]|
        S=Sin, ST_in = ST_out.
alternatively.
    choice(_, _, Sin, S, _, _, ST_in, ST_out):- Sin \= []|
        Sin = S, ST_out = [stop | ST_in].
    choice([{Row,Bits}|Nondets], A, [], S, SBT, PT, ST_in, ST_out) :-
```

```
        vector_element(PT, Row, PTel)|
    choose(Bits, A, [], S, SBT, PT, PTel, Row, 0, Nondets, ST_in, ST_out).
    choice([], A, [], S, _, _, ST_in, ST_out) :- true|
    % No more nondeterminacy; we have the answer.
    S = [A], ST_out = [stop|ST_in].

    choose(_, _, Sin, S, _, _, _, _, _, _, ST_in, ST_out) :-
ST_in=[stop|_]|
        S=Sin, ST_in = ST_out.
alternatively.
    choose(Bits, A, Sin, S, SBT, PT, PTel, Row, Col, Nondets,
           ST_in, ST_out) :- true|
    alt_choose(Bits, A, Sin, S, SBT, PT, PTel, Row, Col, Nondets,
              ST_in, ST_out).

    alt_choose(0, _, Sin, S, _, _, _, _, _, _, ST_in, ST_out) :- true|
    S = Sin, ST_in = ST_out.
otherwise.
    alt_choose(Bits, A, Sin, S, SBT, PT, PTel, Row, Col, Nondets,
              ST_in, ST_out) :-
    LSB := Bits /\ 1,
    LSB = 0 |
    Col1 := Col + 1,
    Bits1 := Bits >> 1,
    choose(Bits1, A, Sin, S, SBT, PT, PTel, Row, Col1, Nondets,
              ST_in, ST_out).
otherwise.
    alt_choose(Bits, A, Sin, S, SBT, PT, PTel, Row, Col, Nondets,
              ST_in, ST_out) :-
    Col1 := Col + 1,
    Bits1 := Bits >> 1,
    vector_element(PTel, Col, PT_Col) |
    check(Nondets, A, Sin, Sin1, SBT, PT, Row, PT_Col, ND, ND, Col,
            ST_in, ST_in1)@priority(*, 2000),
    choose(Bits1, A, Sin1, S, SBT, PT, PTel, Row, Col1, Nondets,
             ST_in1, ST_out)@priority(*, 1000).

    check(_, _, Sin, S, _, _, _, _, _, _, ST_in, ST_out) :-
    ST_in=[stop|_]|       S=Sin, ST_in = ST_out.
alternatively.
    check(L, A, Sin, S, SBT, PT, Row, PT_Cand, Nondets, NDtail,
          Col, ST_in, ST_out) :- true|
    alt_check(L, A, Sin, S, SBT, PT, Row, PT_Cand, Nondets, NDtail,
              Col, ST_in, ST_out).

    alt_check([], A, Sin, S, SBT, PT, ARow, _, Nondets, NDtail,
              Col, ST_in, ST_out) :- true|
    NDtail = [],
    set_vector_element(A, ARow, _, Col, A1),
    choice(Nondets, A1, Sin, S, SBT, PT, ST_in, ST_out).
    alt_check([{NDRow, Bits}|NDleft], A, Sin, S, SBT, PT,
            Row, PT_Cand, Nondets, NDtail, Col, ST_in, ST_out) :-
    vector_element(PT_Cand, NDRow, Pattern),
    NewBits := Bits /\ Pattern,
    NewBits > 0 |
    check1(NDleft, A, Sin, S, SBT, PT, Row, PT_Cand, Nondets, NDtail,
           NewBits, NDRow, Col, ST_in, ST_out).
otherwise.
    alt_check(_, _, Sin, S, _, _, _, _, _, _, _, ST_in, ST_out) :-
```

```
                true|
    S = Sin, ST_in = ST_out.

% to test if the solution is unique
    check1(NDleft, A, Sin, S, SBT, PT, XRow, PT_Cand, Nondets, NDtail,
   Bits, Row, Col, ST_in, ST_out) :-
Hash := Bits mod 37 ,
vector_element(SBT, Hash, {Bits, Pos}) |
check2(NDleft, A, Sin, S, SBT, PT, XRow, PT_Cand, Nondets, NDtail,
       Row, Pos, Col, ST_in, ST_out).
otherwise.
    check1(NDleft, A, Sin, S, SBT, PT, XRow, PT_Cand,
          Nondets, NDtail, Bits, Row, Col, ST_in, ST_out) :-
NDtail = [{Row, Bits}|NewTail],
check(NDleft, A, Sin, S, SBT, PT, XRow, PT_Cand, Nondets, NewTail,
              Col, ST_in, ST_out).

    check2([], A, Sin, S, SBT, PT, ARow, _, Nondets, NDtail, Row, Pos,
           Col, ST_in, ST_out) :-
vector_element(PT, Row, PT_PosI),
vector_element(PT_PosI, Pos, PT_Pos)|
NDtail = [],
% again false
set_vector_element(A, ARow, _, Col, A1),
check(Nondets, A1, Sin, S, SBT, PT, Row, PT_Pos, ND, ND, Pos,
             ST_in, ST_out).
    check2([{NDRow, Bits}|NDleft], A, Sin, S, SBT, PT, XRow, PT_Cand,
   Nondets, NDtail, Row, Pos, Col, ST_in, ST_out) :-
vector_element(PT_Cand, NDRow, Pattern),
NewBits := Bits /\ Pattern,
NewBits > 0 |
NDtail = [{NDRow, NewBits}|NewTail],
check2(NDleft, A, Sin, S, SBT, PT, XRow, PT_Cand,
       Nondets, NewTail, Row, Pos, Col, ST_in, ST_out).
otherwise.
    check2(_, _, Sin, S, _, _, _, _, _, _, _, _, _, ST_in, ST_out) :- true|
S = Sin, ST_in = ST_out.

% INITALIZATIONS Inserted
% ----------------------

% generates a vector of SizeX elements:{Dom1, Dom2, Dom3, Dom4...}
% where Dom represents the bit encoding of each variables domain
% the size of each domain DomSz is recorded in Rsize
% Rsize=[Dom1Sz, Dom1, Dom2Sz, Dom2 ...]
% PT is the mask table (NewDom:= Mask/\ Dom)

init_nondets(SizeX, SizeY, MaxPro, CurPro, R, PT):- true|
      fch:gen_elements(~(SizeY-1), DomM, DomA, DomB, DAsz, DBsz),
      fch:find_power(0, 0, MaxPro, Digit),
      init_nondets1(R, 0, SizeX, DomM, DomA, DomB, DAsz, DBsz, CurPro, Digit,
                    0, PT).

% init_nondets1 fills the domains, It determines as well the parallelism
% According to the number of processors, which is in power of 2,
% it inserts the whole domain of a variable, or inserts a half domain
% DomA or DomB.
% Of course, each processor will work on a different domain
```

```
init_nondets1(St1, Max, Max, _, _, _, _, _, _, _, _):- true|
      St1=[].
otherwise.
init_nondets1(St, N, Max, DomM, DomA, DomB, DAsz, DBsz, CurPro, Digit, Where, PT):-
  N >= Digit,
  vector_element(PT, N, PtEl),
  vector_element(PtEl, 0, PtElEl),
  vector_element(PtElEl, N, InitDom),
  DomEl:= DomM /\ InitDom|
      St=[{N, DomEl}|St1],
      init_nondets1(St1, ~(N+1), Max, DomM, DomA, DomB, DAsz, DBsz, CurPro,
              Digit, Where, PT).
init_nondets1(St, N, Max, DomM, DomA, DomB, DAsz, DBsz, CurPro, Digit, Where, PT):-
  N < Where,
  vector_element(PT, N, PtEl),
  vector_element(PtEl, 0, PtElEl),
  vector_element(PtElEl, N, InitDom),
  DomEl:= DomM /\ InitDom|
      St=[{N, DomEl}|St1],
      init_nondets1(St1, ~(N+1), Max, DomM, DomA, DomB, DAsz, DBsz, CurPro,
              Digit, Where, PT).
init_nondets1(St, N, Max, DomM, DomA, DomB, DAsz, DBsz, CurPro, Digit, Where, PT):-
  N < Digit, N >= Where,
  vector_element(PT, N, PtEl),
  vector_element(PtEl, 0, PtElEl),
  vector_element(PtElEl, N, InitDom)|
      Shift:= N - Where,
      Loc:= (CurPro >>Shift)/\1,
      (Loc=:=1 ->         DomEl:= DomA /\ InitDom,
                          St=[{N, DomEl}|St1];
       otherwise;
       true ->            DomEl:= DomB /\ InitDom,
                          St=[{N, DomEl}|St1]),
      init_nondets1(St1, ~(N+1), Max, DomM, DomA, DomB, DAsz, DBsz, CurPro,
              Digit, Where, PT).
```

3.2 GFCH1

```
% Program:  GENERALIZED FORWARD CHECKING IN PARALLEL
%
%               the size of the tables is a parameter
%
%                     UNIQUE SOLUTION SEARCH
%                  speculative AND-parallelism
%
% Author:  Bernard Burg
% Date:    June 28 90.
%
% Notes:
% run({SizeX, SizeY, WD_encod}, Problem, ProcessorNb, "file")
%
%    SizeX is number of variables in X dimension (integer)
%    SizeY is number of variables in Y dimension (integer)
%    WD_encod defines the length of the encoding and decoding tables. (integer)
%    if WD_encod=1, an element of X is decoded as 1 word
%       WD_encod=2,              X              2 words, with a shift
%       etc.
%    This word encoding has strong repercussions on the efficiency, thus
%    we open this parameter to the user, so he may adapt it to the machine
%    used. (For the 80 MByte PSI, we used SizeX=16, WD_encod=1)
%    Problem is the name of the module containing the application ('atom')
%    ProcessorNb is the number of processors used (integer)
%    File is the name of the result file ("string").
%
%    in the file, time measure, reduction count, Size
%    ProcessorNb and number of solutions are recorded.
%
% This version uses bit encoding for the domains of variables
%
% The pattern masks are stored in the PT table.
% The number of bites of domains are stored in the DT table.
% The first bite of domains are stored in the FT table.
%
% The heart of the program is application independent.


:- module gfch1.
:- public gfch1/5, pgfch1/5, run/5.

%
% PARALLELISM AND MEASURES
% ------------------------

run({VarNbIn, DomSz, WZ}, PrName, OutF, ProNbIn, File):- true|
      fch:fch_input(VarNbIn, VarNb, ProNbIn, ProNb),
      Name=string#"GFCH_ONE",
      fel:get_code(gfch1, pgfch1, 5, Qcode, normal),
      util:start_stats(Qcode, {{VarNb, DomSz, WZ}, PrName, OutF, ProNb, L}, STR),
      STR1=[{0, string#"problem solver", Name},
            {0, string#"problem", PrName},
            {0, string#"size", {VarNb, DomSz, WZ}},
            {0, string#"processors", ProNb},
            {0, string#"solutions", L}|STR],
      (string(File, _, _) -> fch:write_in_file(STR1, File);
       alternatively;
```

```
       true ->          File=L).

pgfch1(N, PrName, OutF, ProNb, Res):- true|
      par:create_list_of_p(0, ~(ProNb-1), {N, PrName, OutF, ProNb}, Data),
      par:apply_list_of_p(Data, _, gfch1, gfch1,
          {data_in, processor_in, merge_out, stream_in, stream_out},
           Res1, Circ, Circ),
      list:sync_sum(Res1, Res2, Tot),
      (wait(Tot) ->
          util:req_time(string#"time", Sync),
          pgfch11(OutF, Tot, Res2, Res, Sync)).

pgfch11(OutF, Tot, Res2, Res, Sync):- wait(Sync)|
          util:req_red(string#"reductions", Sync1),
          (wait(Sync1), OutF=all -> Res=Res2;
           otherwise;
           wait(Sync1) -> Res=[Tot, Res2]).

% the main predicate
% ------------------

gfch1({{SizeX, SizeY, T_SZ}, PrName, OutF, MaxPro},
       CurPro, ME_out, ST_in, ST_out):-
  MMas:= (1<<SizeY)-1|
      gen_dom(PrName, SizeX, SizeY, MaxPro, CurPro, Dom, [Car, Cdar| Lis],
              PT, DT, MasTa, DGNB, Sumln),
      fch:gen_decode(SizeX, SizeY, T_SZ, DT, FT, DGNB, MasTa),
      prop_best(Lis, Car, Cdar, NLis, [NCar, NCdar]),
      fch:sumlength(SizeY, DGNB, Sumln),
      (NCar < 1 -> R1=[], ST_in=ST_out;
       otherwise;
       true ->
        propagate([NCar, NCdar], NLis, Dom, [], R1, DT, PT, FT, DGNB, MasTa,
                  MMas, Sumln, ST_in, ST_out)),
      list:sync_length(R1, R2, LLn),
      wait_res(OutF, LLn, R2, ME_out).

wait_res(all, LLn, R2, R):- wait(LLn)|R=R2.
otherwise.
wait_res(_, LLn, _, R):- wait(LLn)|R=[LLn].

%
% INITALIZATIONS
% --------------

% generates a vector of N elements:{Dom1, Dom2, Dom3, Dom4...}
% where Dom represents the bit encoding of each variables domain
% the size of each domain DomSz is recorded in Rsize
% Rsize=[Dom1Sz, Dom1, Dom2Sz, Dom2 ...]
% PT is the mask table (NewDom:= Mask/\ Dom)
% DT is the table of the number of bites
%    (vector_element(DT, NewDom, NewDomSz))
% FT is the table of the first bites, to find the next
%    instanstanciation of the variables

% T_Size is the size of the words in the decoding
% Mas_TA is the decoding table, it constaints masks of cutting the
% word to decode in slices,
```

```
gen_dom(PrName, SizeX, SizeY, MaxPro, CurPro, R, NonDet, PT, DT, TD,
        DGNB, SumIn):- true|
      fch:gen_elements(~(SizeY-1), DomM, DomA, DomB, DAsz, DBsz),
      fch:find_power(0, 0, MaxPro, Digit),
      Where:= 0,  %(SizeX - Digit)/2, for queens
      (wait(DAsz), wait(Where) ->
        fel:get_code(PrName, constraints, 2, PRCode, normal),
        fel:shoen_execute(PRCode, {{SizeX, SizeY}, Cons}, 0, 4096, 1,
        [start|Ctl], Rpt),
        fch:control_shoen(Rpt, Ctl, SSync),
        fch:mpattern_table(Cons, SizeX, SizeY, PT, SSync, CurPro),
        gen_dom1(R, NonDet, 0, SizeX, SizeY, DomM, DomA, DomB, DAsz, DBsz,
                 CurPro, ~(Where+Digit), Where, PT, DT, TD, DGNB, SumIn)).


% gen_dom1 fills the domains, It determines as well the parallelism
% According to the number of processors, which is in power of 2,
% it inserts the whole domain of a variable, or inserts a half domain
% DomA or DomB.
% Of course, each processor will work on a different domain

gen_dom1(St1, NonDet, SizeX, SizeX, _, _, _, _, _, _, _, _, _, _, _, _, _)
        :- true|
      new_vector(St1, SizeX), NonDet=[].
otherwise.
gen_dom1(St, NonDet, N, SizeX, SizeY, DomM, DomA, DomB, DAsz, DBsz,
        CurPro, Digit, Where, PT, DT, TD, DGNB, SumIn):-
  N >= Digit,
  vector_element(PT, N, PtEl),
  vector_element(PtEl, 0, PtElEl),
  vector_element(PtElEl, N, InitDom),
  DomEl:= DomM /\ InitDom|
      set_vector_element(St1, N, _, DomEl, St),
      bit_sum(DT, DomEl, SumIn, TD, DGNB, DomLn, 0),
      NonDet=[DomLn, N|NonDet1],
      gen_dom1(St1, NonDet1, ~(N+1), SizeX, SizeY, DomM, DomA, DomB, DAsz,
               DBsz, CurPro, Digit, Where, PT, DT, TD, DGNB, SumIn).
gen_dom1(St, NonDet, N, SizeX, SizeY, DomM, DomA, DomB, DAsz, DBsz,
        CurPro, Digit, Where, PT, DT, TD, DGNB, SumIn):-
  N < Where,
  vector_element(PT, N, PtEl),
  vector_element(PtEl, 0, PtElEl),
  vector_element(PtElEl, N, InitDom),
  DomEl:= DomM /\ InitDom|
      set_vector_element(St1, N, _, DomEl, St),
      bit_sum(DT, DomEl, SumIn, TD, DGNB, DomLn, 0),
      NonDet=[DomLn, N|NonDet1],
      gen_dom1(St1, NonDet1, ~(N+1), SizeX, SizeY, DomM, DomA, DomB,
               DAsz, DBsz, CurPro, Digit, Where, PT, DT, TD, DGNB, SumIn).
gen_dom1(St, NonDet, N, SizeX, SizeY, DomM, DomA, DomB, DAsz, DBsz,
        CurPro, Digit, Where, PT, DT, TD, DGNB, SumIn):-
  N < Digit, N >= Where,
  Shift:= N - Where,
  Loc:= (CurPro >>Shift)/\1,
  vector_element(PT, N, PtEl),
  vector_element(PtEl, 0, PtElEl),
  vector_element(PtElEl, N, InitDom)|
      (Loc=:=1 ->       DomEl:= DomA /\ InitDom,
                        set_vector_element(St1, N, _, DomEl, St);
```

```
      otherwise;
      true ->          DomEl:= DomB /\ InitDom,
                       set_vector_element(St1, N, _, DomEl, St)),
     bit_sum(DT, DomEl, Sumln, TD, DGNB, DomLn, 0),
     NonDet=[DomLn, N|NonDet1],
     gen_dom1(St1, NonDet1, ~(N+1), SizeX, SizeY, DomM, DomA, DomB,
              DAsz, DBsz, CurPro, Digit, Where, PT, DT, TD, DGNB, Sumln).


% GENERALIZED FORWARD CHECKING (core)
% ----------------------------------

% a solution was given by another processor.
propagate(_, _, _, In, Out, _, _, _, _, _, _, _, ST_in, ST_out):-
  ST_in=[stop|_]|ST_out=ST_in, Out=In.
alternatively.
propagate(_, _, _, In, Out, _, _, _, _, _, _, _, ST_in, ST_out):-
  In \= []| Out=In, ST_out=[stop|ST_in].
propagate([], _, Dom, [], Out, _, _, _, _, _, _, _, ST_in, ST_out):-
          true|Out=[Dom], ST_out=[stop|ST_in].
propagate([1,L], Lis, Dom, [], Out, Dt, Pt, Ft, DGNB, TD, MMas, Sumln, ST_in,
          ST_out):-
  vector_element(Pt, L, Ptel)|
       set_vector_element(Dom, L, Doml, Y, Domc),
       first_bit(Ft, Doml, 0, TD, DGNB, Y),
       vector_element(Ptel, Y, Ptelel, _),
       check_pos1(Lis, Domc, [], [], Out, Ptelel, Dt, Pt, Ft, DGNB, TD,
                  MMas, Sumln,ST_in, ST_out).
propagate([Lln,L], Lis, Dom, [], Out, Dt, Pt, Ft, DGNB, TD, MMas, Sumln,
          ST_in, ST_out):-
  Lln=\=1,
  vector_element(Pt, L, Ptel)|
       set_vector_element(Dom, L, Doml, Doml2, Domc),
       first_bit(Ft, Doml, 0, TD, DGNB, Y),
       Y1:=Y+1,
       Doml2:=(Doml>>Y1)<<Y1,
       set_vector_element(Domc, L, _, Y, Domc3),
       vector_element(Ptel, Y, Ptelel, _),
       check_pos2(Lis, Domc3, [], [~(Lln-1),L|Lis], Domc, [], Out, Ptelel, Dt,
                  Pt, Ft, DGNB, TD, MMas, Sumln, ST_in, ST_out).


% propagates The Constraints Issued From Position [X, Y], For Line L.
check_pos1( _, _, _, In, Out, _, _, _, _, _, _, _, ST_in, ST_out):-
  ST_in=[stop|_]|ST_out=ST_in, Out=In.
alternatively.
check_pos1(L, Dom, Varr, In, Out, Ptel, Dt, Pt, Ft, DGNB, TD, MMas,
           Sumln, ST_in, ST_out):- true|
    alt_check_pos1(L, Dom, Varr, In, Out, Ptel, Dt, Pt, Ft, DGNB, TD,
           MMas, Sumln, ST_in, ST_out).

alt_check_pos1( _, _, [0|_], In, Out, _, _, _, _, _, _, _, _,
                ST_in, ST_out):- true|
      Out=In, ST_in=ST_out.
otherwise.
alt_check_pos1([Aln,A|B], Dom, Varr, In, Out, Ptel, Dt, Pt, Ft,
                DGNB, TD, MMas, Sumln, ST_in, ST_out):-
  vector_element(Ptel, A, MMas)|
      check_pos1(B, Dom, [Aln, A |Varr], In, Out, Ptel, Dt, Pt, Ft,
```

```
                DGNB, TD, MMas, Sumln, ST_in, ST_out).
otherwise.
alt_check_pos1([_,A|B], Dom, Varr, In, Out, Ptel, Dt, Pt, Ft, DGNB,
               TD, MMas, Sumln, ST_in, ST_out):-
  vector_element(Ptel, A, Mask)|
      set_vector_element(Dom, A, Domel, Newdom, Dom1),
      Newdom:= Domel /\ Mask,
      bit_sum(Dt, Newdom, Sumln, TD, DGNB, Newaln, 0),
      check_pos1(B, Dom1, [Newaln, A |Varr], In, Out, Ptel, Dt, Pt, Ft, DGNB,
                 TD, MMas, Sumln, ST_in, ST_out).
alt_check_pos1([], Dom, [], In, Out, _, _, _, _, _, _, _, _,
               ST_in, ST_out):- true|
      Out=[Dom|In], ST_out=[stop|ST_in].
alt_check_pos1([], Dom, [Car, Cadr|Cdr], In, Out, _, Dt, Pt, Ft,
               DGNB, TD, MMas, Sumln, ST_in, ST_out):- true|
      lazy_prop_best(Cdr, Car, Cadr, Ncdr, Ncar),
      propagate(Ncar, Ncdr, Dom, In, Out, Dt, Pt, Ft, DGNB, TD, MMas,
                Sumln, ST_in, ST_out).

check_pos2( _, _, _, _, _, In, Out, _, _, _, _, _, _, _, _,
          ST_in, ST_out):-
  ST_in=[stop|_]|ST_out=ST_in, Out=In.
alternatively.
check_pos2(L, Dom, VarR, ValIn, DomIn, In, Out, PTEL, DT, PT, FT,
          DGNB, TD, MMas, Sumln, ST_in, ST_out):- true|
    alt_check_pos2(L, Dom, VarR, ValIn, DomIn, In, Out, PTEL, DT,
                   PT, FT, DGNB, TD, MMas, Sumln, ST_in, ST_out).

alt_check_pos2( _, _, [0|_], [Lln, L|Lis], DomIn, In, Out, _, DT, PT,
          FT, DGNB, TD, MMas, Sumln, ST_in, ST_out):- true|
      propagate([Lln,L], Lis, DomIn, In, Out, DT, PT, FT, DGNB, TD,
                MMas, Sumln, ST_in, ST_out).
otherwise.
alt_check_pos2([Aln,A|B], Dom, VarR, ValIn, DomIn, In, Out, PTEL,
               DT, PT, FT, DGNB, TD, MMas, Sumln, ST_in, ST_out):-
  vector_element(PTEL, A, MMas)|
      check_pos2(B, Dom, [Aln, A |VarR], ValIn, DomIn, In, Out, PTEL,
                 DT, PT, FT, DGNB, TD, MMas, Sumln, ST_in, ST_out).
otherwise.
alt_check_pos2([_,A|B], Dom, VarR, ValIn, DomIn, In, Out, PTEL, DT,
          PT, FT, DGNB, TD, MMas, Sumln, ST_in, ST_out):-
  vector_element(PTEL, A, Mask)|
      set_vector_element(Dom, A, DomEL, NewDom, Dom1),
      NewDom:= DomEL /\ Mask,
      bit_sum(DT, NewDom, Sumln, TD, DGNB, NewAln, 0),
      check_pos2(B, Dom1, [NewAln, A |VarR], ValIn, DomIn, In, Out,
                 PTEL, DT, PT, FT, DGNB, TD, MMas, Sumln, ST_in, ST_out).
alt_check_pos2([], Dom, [], _, _, In, Out, _, _, _, _, _, _, _, _,
               ST_in, ST_out):- true|
      Out=[Dom|In], ST_out=[stop|ST_in].
alt_check_pos2([], Dom, [Car, Cadr|Cdr], [Lln, L|Lis], DomIn, In,
               Out, _, DT, PT, FT, DGNB, TD, MMas, Sumln, ST_in, ST_out):- true|
      propagate([Lln,L], Lis, DomIn, In, In1, DT, PT, FT, DGNB, TD, MMas,
                Sumln, ST_in, ST_in1)@priority(*, 2000),
      lazy_prop_best(Cdr, Car, Cadr, NCdr, NCar),
      propagate(NCar, NCdr, Dom, In1, Out, DT, PT, FT, DGNB, TD, MMas,
                Sumln, ST_in1, ST_out)@priority(*, 1000).
```

```
% the following predicates are used in the core of the program
% they should be as efficient as possible
% note: they are shared by GFCH and GFCH1, but for the sake of
% efficiency we duplicate this code in the modules calling them.

% research of the first bite of a word
first_bit(Ft, Value, Pos, BMask, DGNB, Res):-
  Index:= BMask /\ Value|
      first_bit1(Index, Ft, Value, Pos, BMask, DGNB, Res).

first_bit1(0, Ft, Value, Pos, BMask, DGNB, Res):-
  NValue := Value >> DGNB|
      first_bit(Ft, NValue, ~(Pos+1), BMask, DGNB, Res).
first_bit1(Index, Ft, _, Pos, _, DGNB, Res):-
  Index > 0,
  vector_element(Ft, Index, Loc)|
      Res:= Loc + (Pos * DGNB).

% research of the sum of the bites of a word
bit_sum(Dt, Value, Pos, BMask, DGNB, Res, Sum):-
  Pos > 0,
  NValue := Value >> DGNB,
  Index := BMask /\ Value,
  vector_element(Dt, Index, RLoc)|
      Sum1:= RLoc + Sum,
      bit_sum(Dt, NValue, ~(Pos-1), BMask, DGNB, Res, Sum1).
bit_sum(Dt, Value, 0, _, _, Res, Sum):-
  vector_element(Dt, Value, RLoc)|
      Res:=RLoc+Sum.


% searches for the best element to be propagated,
% it means the one having the smallest Domain Size DomSz,
% [DomSz, Dom |others]

prop_best([A,B|L], Min, C, Res, Couple):- A > Min|
      Res=[A,B|Res1],
      prop_best(L, Min, C, Res1, Couple).
prop_best([A,B|L], Min, C, Res, Couple):- A =< Min|
      Res=[Min, C|Res1],
      prop_best(L, A, B, Res1, Couple).
prop_best([], A, B, Res, Couple):- true|
      Res=[], [A, B]=Couple.


% the Lazy version is used in the core of the program
% searches for the best element to be propagated,
% it means the one having the smallest Domain Size DomSz,
% [DomSz, Dom |others]
% this lazy version stops as soon as it finds and element having
% a Domain size=1, [1, Dom, |others]. This works since we know that
% 0 elements cannot appear in the core of the program

lazy_prop_best(L, 1, C, Res, Couple):- true|
      Res=L, Couple=[1,C].
```

```
otherwise.
lazy_prop_best([A,B|L], Min, C, Res, Couple):- A > Min|
      Res=[A,B|Res1],
      lazy_prop_best1(L, Min, C, Res1, Couple).
lazy_prop_best([A,B|L], Min, C, Res, Couple):- A =< Min|
      Res=[Min, C|Res1],
      lazy_prop_best(L, A, B, Res1, Couple).
lazy_prop_best([], A, B, Res, Couple):- true|
      Res=[], [A, B]=Couple.

lazy_prop_best1([A,B|L], Min, C, Res, Couple):- A > Min|
      Res=[A,B|Res1],
      lazy_prop_best1(L, Min, C, Res1, Couple).
lazy_prop_best1([A,B|L], Min, C, Res, Couple):- A =< Min|
      Res=[Min, C|Res1],
      lazy_prop_best(L, A, B, Res1, Couple).
lazy_prop_best1([], A, B, Res, Couple):- true|
      Res=[], [A, B]=Couple.
```

3.3 SFCH

```
% Program:    SIMPLE FORWARD CHECKING IN PARALLEL
%
%                exhaustive solution search
%
%
%   Author:     Takashi Chikayama (ICOT)
%   Date:       15 Aug 1988
%   Note:       Queens Problem Solver
%               Originally, this was a sequential implementation of the
%               queen prolem
%
%   modified    Bernard Burg
%               Extention to parallel processing
%   Date:       10 May 90.

%   modified    Bernard Burg
%               Extension to simple forward checking (application indpt)
%   Date:       13 June 90.

%
%   For keeping candidate positions, a bit table is used.
%   Thus, only up to 32 variables problem can be solved.
%


:- module sfch.

:- public main/3, run/5, psfch/5.

% measures for the multi-psi
run({VarNbIn, DomSZ}, PrName, OutF, ProNbIn, File):- true|
      fch:fch_input(VarNbIn, VarNb, ProNbIn, ProNb),
      Name=string#"SFCH",
      fel:get_code(sfch, psfch, 5, Qcode, normal),
      util:start_stats(Qcode, {{VarNb, DomSZ}, PrName, OutF, ProNb, L}, STR),
      STR1=[{0, string#"problem solver", Name},
            {0, string#"problem", PrName},
            {0, string#"size", {VarNb, DomSZ}},
            {0, string#"processors", ProNb},
            {0, string#"solutions", L}|STR],
     (string(File, _, _) ->  fch:write_in_file(STR1, File);
      alternatively;
      true -> File=L).

psfch(N, PrName, OutF, ProNb, Res):- true|
      par:create_list_of_p(0, ~(ProNb-1), {N, PrName, OutF, ProNb}, Data),
      par:apply_list_of_p(Data, _, sfch, main,
         {data_in, processor_in, merge_out}, Res1, _, []),
      list:sync_sum(Res1, Res2, Tot),
      (wait(Tot) ->
          util:req_time(string#"time", Sync),
          psfch1(OutF, Tot, Res2, Res, Sync)).

psfch1(OutF, Tot, Res2, Res, Sync):- wait(Sync)|
          util:req_red(string#"reductions", Sync1),
          (wait(Sync1), OutF=all -> Res=Res2;
           otherwise;
```

```
            true ->     Res=[Tot, Res2]).

 main({Size, PrName, OutF, MaxPro}, CurPro, R) :- true|
       main1({Size, PrName, MaxPro}, CurPro, Ans),
       list:sync_length(Ans, AnsC, LLn),
       (wait(LLn), OutF=all -> R=AnsC;
        otherwise;
       true ->   R=[LLn]).


     main1({{SizeX, SizeY}, PrName, MaxPro}, CurPro, Ans) :-
     init_nondets(SizeX, SizeY, MaxPro, CurPro, Nondets, PT),
     fch:single_bit_table(SBT),
     fel:get_code(PrName, constraints, 2, PRCode, normal),
     fel:shoen_execute(PRCode, {{SizeX, SizeY}, Cons}, 0, 4096, 1,
     [start|Ctl], Rpt),
     fch:control_shoen(Rpt, Ctl, SSync),
     fch:mpattern_table(Cons, SizeX, SizeY, PT, SSync, CurPro),
     merge(S, Ans),
     new_vector(AnsV, SizeX),
     choice(Nondets, AnsV, S, SBT, PT).

     choice([{Row,Bits}|Nondets], A, S, SBT, PT) :-
         vector_element(PT, Row, PTel)|
 choose(Bits, A, S, SBT, PT, PTel, Row, 0, Nondets).
     choice([], A, S, _, _) :- true|
 % No more nondeterminacy; we have the answer.
 S = [A].

     choose(0, _, S, _, _, _, _, _, _) :- true|
 S = [].
 otherwise.
     choose(Bits, A, S, SBT, PT, PTel, Row, Col, Nondets) :-
 LSB := Bits /\ 1,
 LSB = 0 |
 Col1 := Col + 1,
 Bits1 := Bits >> 1,
 choose(Bits1, A, S, SBT, PT, PTel, Row, Col1, Nondets).
 otherwise.
     choose(Bits, A, S, SBT, PT, PTel, Row, Col, Nondets) :-
 Col1 := Col + 1,
 Bits1 := Bits >> 1,
 vector_element(PTel, Col, PT_Col) |
 S = {S1, S2},
 check(Nondets, A, S1, SBT, PT, Row, PT_Col, ND, ND, Col)@priority(*, 2000),
 choose(Bits1, A, S2, SBT, PT, PTel, Row, Col1, Nondets)@priority(*, 1000).

     check([], A, S, SBT, PT, ARow, _, Nondets, NDtail, Col) :-
     true|
 NDtail = [],
 set_vector_element(A, ARow, _, Col, A1),
 choice(Nondets, A1, S, SBT, PT).
     check([{NDRow, Bits}|NDleft], A, S, SBT, PT,
   Row, PT_Cand, Nondets, NDtail, Col) :-
 vector_element(PT_Cand, NDRow, Pattern),
 NewBits := Bits /\ Pattern,
 NewBits > 0 |
 check1(NDleft, A, S, SBT, PT, Row, PT_Cand, Nondets, NDtail,
        NewBits, NDRow, Col).
```

```
otherwise.
    check(_, _, S, _, _, _, _, _, _, _) :- true|
S = [].

% to test if the solution is unique
    check1(NDleft, A, S, SBT, PT, XRow, PT_Cand, Nondets, NDtail,
   Bits, Row, Col) :-
Hash := Bits mod 37 ,
vector_element(SBT, Hash, {Bits, Pos}) |
check2(NDleft, A, S, SBT, PT, XRow, PT_Cand, Nondets, NDtail,
       Row, Pos, Col).
otherwise.
    check1(NDleft, A, S, SBT, PT, XRow, PT_Cand,
          Nondets, NDtail, Bits, Row, Col) :-
NDtail = [{Row, Bits}|NewTail],
check(NDleft, A, S, SBT, PT, XRow, PT_Cand, Nondets, NewTail, Col).

    check2([], A, S, SBT, PT, ARow, _, Nondets, NDtail, Row, Pos, Col) :-
vector_element(PT, Row, PT_PosI),
vector_element(PT_PosI, Pos, PT_Pos)|
NDtail = [],
% again false
set_vector_element(A, ARow, _, Col, A1),
check(Nondets, A1, S, SBT, PT, Row, PT_Pos, ND, ND, Pos).
    check2([{NDRow, Bits}|NDleft], A, S, SBT, PT, XRow, PT_Cand,
   Nondets, NDtail, Row, Pos, Col) :-
vector_element(PT_Cand, NDRow, Pattern),
NewBits := Bits /\ Pattern,
NewBits > 0 |
NDtail = [{NDRow, NewBits}|NewTail],
check2(NDleft, A, S, SBT, PT, XRow, PT_Cand,
       Nondets, NewTail, Row, Pos, Col).
otherwise.
    check2(_, _, S, _, _, _, _, _, _, _, _, _) :- true|
S = [].

% INITALIZATIONS  Inserted
% ------------------------

% generates a vector of SizeX elements:{Dom1, Dom2, Dom3, Dom4...}
% where Dom represents the bit encoding of each variables domain
% the size of each domain DomSz is recorded in Rsize
% Rsize=[Dom1Sz, Dom1, Dom2Sz, Dom2 ...]
% PT is the mask table (NewDom:= Mask/\ Dom)

init_nondets(SizeX, SizeY, MaxPro, CurPro, R, PT):- true|
      fch:gen_elements(~(SizeY-1), DomM, DomA, DomB, DAsz, DBsz),
      fch:find_power(0, 0, MaxPro, Digit),
      init_nondets1(R, 0, SizeX, DomM, DomA, DomB, DAsz, DBsz, CurPro, Digit,
                    0, PT).

% init_nondets1 fills the domains, It determines as well the parallelism
% According to the number of processors, which is in power of 2,
% it inserts the whole domain of a variable, or inserts a half domain
% DomA or DomB.
% Of course, each processor will work on a different domain

init_nondets1(St1, Max, Max, _, _, _, _, _, _, _, _, _):- true|
      St1=[].
```

```
otherwise.
init_nondets1(St, N, Max, DomM, DomA, DomB, DAsz, DBsz, CurPro, Digit, Where, PT):-
  N >= Digit,
  vector_element(PT, N, PtEl),
  vector_element(PtEl, 0, PtElEl),
  vector_element(PtElEl, N, InitDom),
  DomEl:= DomM /\ InitDom|
      St=[{N, DomEl}|St1],
      init_nondets1(St1, ~(N+1), Max, DomM, DomA, DomB, DAsz, DBsz, CurPro,
              Digit, Where, PT).
init_nondets1(St, N, Max, DomM, DomA, DomB, DAsz, DBsz, CurPro, Digit, Where, PT):-
  N < Where,
  vector_element(PT, N, PtEl),
  vector_element(PtEl, 0, PtElEl),
  vector_element(PtElEl, N, InitDom),
  DomEl:= DomM /\ InitDom|
      St=[{N, DomEl}|St1],
      init_nondets1(St1, ~(N+1), Max, DomM, DomA, DomB, DAsz, DBsz, CurPro,
              Digit, Where, PT).
init_nondets1(St, N, Max, DomM, DomA, DomB, DAsz, DBsz, CurPro, Digit, Where, PT):-
  N < Digit, N >= Where,
  vector_element(PT, N, PtEl),
  vector_element(PtEl, 0, PtElEl),
  vector_element(PtElEl, N, InitDom)|
      Shift:= N - Where,
      Loc:= (CurPro >>Shift)/\1,
      (Loc=:=1 ->         DomEl:= DomA /\ InitDom,
                          St=[{N, DomEl}|St1];
       otherwise;
       true ->            DomEl:= DomB /\ InitDom,
                          St=[{N, DomEl}|St1]),
      init_nondets1(St1, ~(N+1), Max, DomM, DomA, DomB, DAsz, DBsz, CurPro,
              Digit, Where, PT).
```

3.4 GFCH

```
% Program:  GENERALIZED FORWARD CHECKING IN PARALLEL
%
%               the size of the tables is a parameter
%                 exhaustive solution search
%
%
% Author:  Bernard Burg
% Date:    June 14 90.
%
% Notes:
% run({SizeX, SizeY, WD_encod}, Problem, ProcessorNb, "file")
%
%    SizeX is number of variables in X dimension (integer)
%    SizeY is number of variables in Y dimension (integer)
%    WD_encod defines the length of the encoding and decoding tables. (integer)
%    if WD_encod=1, an element of X is decoded as 1 word
%       WD_encod=2,               X               2 words, with a shift
%       etc.
%    This word encoding has strong repercussions on the efficiency, thus
%    we open this parameter to the user, so he may adapt it to the machine
%    used. (For the 80 MByte PSI, we used SizeX=16, WD_encod=1)
%    Problem is the name of the module containing the application ('atom')
%    ProcessorNb is the number of processors used (integer)
%    File is the name of the result file ("string").
%
%    in the file, time measure, reduction count, Size
%    ProcessorNb and number of solutions are recorded.
%
% This version uses bit encoding for the domains of variables
%
% The pattern masks are stored in the PT table.
% The number of bites of domains are stored in the DT table.
% The first bite of domains are stored in the FT table.
%


:- module gfch.
:- public gfch/3, pgfch/5, run/5.

%
% PARALLELISM AND MEASURES
% ------------------------

run({VarNbIn, DomSz, WZ}, PrName, OutF, ProNbIn, File):- true|
      fch:fch_input(VarNbIn, VarNb, ProNbIn, ProNb),
      Name=string#"GFCH",
      fel:get_code(gfch, pgfch, 5, Qcode, normal),
      util:start_stats(Qcode, {{VarNb, DomSz, WZ}, PrName, OutF, ProNb, L},
                       STR),
      STR1=[{0, string#"problem solver", Name},
            {0, string#"problem", PrName},
            {0, string#"size", {VarNb, DomSz, WZ}},
            {0, string#"processors", ProNb},
            {0, string#"solutions", L}|STR],
      (string(File, _, _) -> fch:write_in_file(STR1, File);
       alternatively;
       true ->          File=L).
```

```
pgfch(N, PrName, OutF, ProNb, Res):- true|
      par:create_list_of_p(0, ~(ProNb-1), {N, PrName, OutF, ProNb}, Data),
      par:apply_list_of_p(Data, _, gfch, gfch,
          {data_in, processor_in, merge_out}, Res1, _, []),
      list:sync_sum(Res1, Res2, Tot),
      (wait(Tot) ->
          util:req_time(string#"time", Sync),
          pgfch1(OutF, Tot, Res2, Res, Sync)).

pgfch1(OutF, Tot, Res2, Res, Sync):- wait(Sync)|
          util:req_red(string#"reductions", Sync1),
          (wait(Sync1), OutF=all -> Res=Res2;
           otherwise;
           wait(Sync1) -> Res=[Tot, Res2]).

% the main predicate
% ------------------

gfch({{SizeX, SizeY, T_SZ}, PrName, OutF, MaxPro},  CurPro, R):-
  MMas:= (1<<SizeY)-1|
      gen_dom(PrName, SizeX, SizeY, MaxPro, CurPro, Dom, [Car, Cdar| Lis],
              PT, DT, MasTa, DGNB, Sumln),
      fch:gen_decode(SizeX, SizeY, T_SZ, DT, FT, DGNB, MasTa),
      prop_best(Lis, Car, Cdar, NLis, [NCar, NCdar]),
      fch:sumlength(SizeY, DGNB, Sumln),
      (NCar < 1 -> R1 =[];
       otherwise;
       true -> propagate([NCar, NCdar], NLis, Dom, R1, DT, PT, FT, DGNB,
                         MasTa, MMas, Sumln)),
      list:sync_length(R1, R2, LLn),
      wait_res(OutF, LLn, R2, R).


wait_res(all, LLn, R2, R):- wait(LLn)|R=R2.
otherwise.
wait_res(_, LLn, _, R):- wait(LLn)|R=[LLn].

%
% INITALIZATIONS
% --------------

% generates a vector of N elements:{Dom1, Dom2, Dom3, Dom4...}
% where Dom represents the bit encoding of each variables domain
% the size of each domain DomSz is recorded in Rsize
% Rsize=[Dom1Sz, Dom1, Dom2Sz, Dom2 ...]
% PT is the mask table (NewDom:= Mask/\ Dom)
% DT is the table of the number of bites
%    (vector_element(DT, NewDom, NewDomSz))
% FT is the table of the first bites, to find the next
%    instanstanciation of the variables

% T_Size is the size of the words in the decoding
% Mas_TA is the decoding table, it constaints masks of cutting the
% word to decode in slices,

gen_dom(PrName, SizeX, SizeY, MaxPro, CurPro, R, NonDet, PT, DT, TD,
        DGNB, Sumln):- true|
      fch:gen_elements(~(SizeY-1), DomM, DomA, DomB, DAsz, DBsz),
```

```
      fch:find_power(0, 0, MaxPro, Digit),
      Where:= 0,  %(SizeX - Digit)/2, for queens
      (wait(DAsz), wait(Where) ->
        fel:get_code(PrName, constraints, 2, PRCode, normal),
        fel:shoen_execute(PRCode, {{SizeX, SizeY}, Cons}, 0, 4096, 1,
        [start|Ctl], Rpt),
        fch:control_shoen(Rpt, Ctl, SSync),
        fch:mpattern_table(Cons, SizeX, SizeY, PT, SSync, CurPro),
        gen_dom1(R, NonDet, 0, SizeX, SizeY, DomM, DomA, DomB, DAsz, DBsz,
                 CurPro, ~(Where+Digit), Where, PT, DT, TD, DGNB, SumIn)).


% gen_dom1 fills the domains, It determines as well the parallelism
% According to the number of processors, which is in power of 2,
% it inserts the whole domain of a variable, or inserts a half domain
% DomA or DomB.
% Of course, each processor will work on a different domain

gen_dom1(St1, NonDet, SizeX, SizeX, _, _, _, _, _, _, _, _, _, _, _, _, _)
         :- true|
       new_vector(St1, SizeX), NonDet=[].
otherwise.
gen_dom1(St, NonDet, N, SizeX, SizeY, DomM, DomA, DomB, DAsz, DBsz,
         CurPro, Digit, Where, PT, DT, TD, DGNB, SumIn):-
  N >= Digit,
  vector_element(PT, N, PtEl),
  vector_element(PtEl, 0, PtElEl),
  vector_element(PtElEl, N, InitDom),
  DomEl:= DomM /\ InitDom|
      set_vector_element(St1, N, _, DomEl, St),
      bit_sum(DT, DomEl, SumIn, TD, DGNB, DomLn, 0),
      NonDet=[DomLn, N|NonDet1],
      gen_dom1(St1, NonDet1, ~(N+1), SizeX, SizeY, DomM, DomA, DomB, DAsz,
               DBsz, CurPro, Digit, Where, PT, DT, TD, DGNB, SumIn).
gen_dom1(St, NonDet, N, SizeX, SizeY, DomM, DomA, DomB, DAsz, DBsz,
         CurPro, Digit, Where, PT, DT, TD, DGNB, SumIn):-
  N < Where,
  vector_element(PT, N, PtEl),
  vector_element(PtEl, 0, PtElEl),
  vector_element(PtElEl, N, InitDom),
  DomEl:= DomM /\ InitDom|
      set_vector_element(St1, N, _, DomEl, St),
      bit_sum(DT, DomEl, SumIn, TD, DGNB, DomLn, 0),
      NonDet=[DomLn, N|NonDet1],
      gen_dom1(St1, NonDet1, ~(N+1), SizeX, SizeY, DomM, DomA, DomB,
               DAsz, DBsz, CurPro, Digit, Where, PT, DT, TD, DGNB, SumIn).
gen_dom1(St, NonDet, N, SizeX, SizeY, DomM, DomA, DomB, DAsz, DBsz,
         CurPro, Digit, Where, PT, DT, TD, DGNB, SumIn):-
  N < Digit, N >= Where,
  Shift:= N - Where,
  Loc:= (CurPro >>Shift)/\1,
  vector_element(PT, N, PtEl),
  vector_element(PtEl, 0, PtElEl),
  vector_element(PtElEl, N, InitDom)|
      (Loc=:=1 ->       DomEl:= DomA /\ InitDom,
                        set_vector_element(St1, N, _, DomEl, St);
       otherwise;
       true ->          DomEl:= DomB /\ InitDom,
                        set_vector_element(St1, N, _, DomEl, St)),
```

```
        bit_sum(DT, DomEl, Sumln, TD, DGNB, DomLn, 0),
        NonDet=[DomLn, N|NonDet1],
        gen_dom1(St1, NonDet1, ~(N+1), SizeX, SizeY, DomM, DomA, DomB,
                 DAsz, DBsz, CurPro, Digit, Where, PT, DT, TD, DGNB, Sumln).


% GENERALIZED FORWARD CHECKING (core)
% ----------------------------------


propagate([], _, Dom, Out, _, _, _, _, _, _, _):- true|Out=[Dom].
propagate([1,L], Lis, Dom, Out, Dt, Pt, Ft, DGNB, TD, MMas, Sumln):-
  vector_element(Pt, L, Ptel)|
        set_vector_element(Dom, L, Doml, Y, Domc),
        first_bit(Ft, Doml, 0, TD, DGNB, Y),
        vector_element(Ptel, Y, Ptelel, _),
        check_pos1(Lis, Domc, [], Out, Ptelel, Dt, Pt, Ft, DGNB, TD, MMas, Sumln).
otherwise.
propagate([Lln,L], Lis, Dom, Out, Dt, Pt, Ft, DGNB, TD, MMas, Sumln):-
  vector_element(Pt, L, Ptel)|
        set_vector_element(Dom, L, Doml, Doml2, Domc),
        first_bit(Ft, Doml, 0, TD, DGNB, Y),
        Y1:=Y+1,
        Doml2:=(Doml>>Y1)<<Y1,
        set_vector_element(Domc, L, _, Y, Domc3),
        vector_element(Ptel, Y, Ptelel, _),
        check_pos2(Lis, Domc3, [], [~(Lln-1),L|Lis], Domc, Out, Ptelel, Dt,
                   Pt, Ft, DGNB, TD, MMas, Sumln).

% propagates The Constraints Issued From Position [X, Y], For Line L.

check_pos1( _, _, [0|_], Out, _, _, _, _, _, _, _, _):- true|
      Out=[].
otherwise.
check_pos1([Aln,A|B], Dom, Varr, Out, Ptel, Dt, Pt, Ft, DGNB, TD, MMas,
           Sumln):-
  vector_element(Ptel, A, MMas)|
      check_pos1(B, Dom, [Aln, A |Varr], Out, Ptel, Dt, Pt, Ft, DGNB, TD,
                 MMas, Sumln).
otherwise.
check_pos1([_,A|B], Dom, Varr, Out, Ptel, Dt, Pt, Ft, DGNB, TD, MMas, Sumln):-
  vector_element(Ptel, A, Mask)|
      set_vector_element(Dom, A, Domel, Newdom, Dom1),
      Newdom:= Domel /\ Mask,
      bit_sum(Dt, Newdom, Sumln, TD, DGNB, Newaln, 0),
      check_pos1(B, Dom1, [Newaln, A |Varr], Out, Ptel, Dt, Pt, Ft, DGNB,
                 TD, MMas, Sumln).
check_pos1([], Dom, [], Out, _, _, _, _, _, _, _, _):- true|
      Out=[Dom].
check_pos1([], Dom, [Car, Cadr|Cdr], Out, _, Dt, Pt, Ft, DGNB, TD, MMas, Sumln):-
  true|
      lazy_prop_best(Cdr, Car, Cadr, Ncdr, Ncar),
      propagate(Ncar, Ncdr, Dom, Out, Dt, Pt, Ft, DGNB, TD, MMas, Sumln).

check_pos2( _, _, [0|_], [Lln, L|Lis], DomIn, Out, _, DT, PT, FT, DGNB,
          TD, MMas, Sumln):- true|
      propagate([Lln,L], Lis, DomIn, Out, DT, PT, FT, DGNB, TD, MMas, Sumln).
otherwise.
check_pos2([Aln,A|B], Dom, VarR, ValIn, DomIn, Out, PTEL, DT, PT, FT, DGNB,
```

```
          TD, MMas, SumIn):-
  vector_element(PTEL, A, MMas)|
      check_pos2(B, Dom, [AIn, A |VarR], ValIn, DomIn, Out, PTEL, DT, PT,
                 FT, DGNB, TD, MMas, SumIn).
otherwise.
check_pos2([_,A|B], Dom, VarR, ValIn, DomIn, Out, PTEL, DT, PT, FT, DGNB,
          TD, MMas, SumIn):-
  vector_element(PTEL, A, Mask)|
      set_vector_element(Dom, A, DomEL, NewDom, Dom1),
      NewDom:= DomEL /\ Mask,
      bit_sum(DT, NewDom, SumIn, TD, DGNB, NewAln, 0),
      check_pos2(B, Dom1, [NewAln, A |VarR], ValIn, DomIn, Out, PTEL, DT,
                 PT, FT, DGNB, TD, MMas, SumIn).
check_pos2([], Dom, [], [LIn, L|Lis], DomIn, Out, _, DT, PT, FT, DGNB,
          TD, MMas, SumIn):- true|
      merge({[Dom], Out1}, Out),
      propagate([LIn,L], Lis, DomIn, Out1, DT, PT, FT, DGNB, TD, MMas, SumIn).
check_pos2([], Dom, [Car, Cadr|Cdr], [LIn, L|Lis], DomIn, Out, _, DT, PT,
           FT, DGNB, TD, MMas, SumIn):- true|
      merge({Out1, Out2}, Out),
      propagate([LIn,L], Lis, DomIn, Out1, DT, PT, FT, DGNB, TD, MMas, SumIn)
                @priority(*, 2000),
      lazy_prop_best(Cdr, Car, Cadr, NCdr, NCar),
      propagate(NCar, NCdr, Dom, Out2, DT, PT, FT, DGNB, TD, MMas, SumIn)
                @priority(*, 1000).



% the following predicates are used in the core of the program
% they should be as efficient as possible
% note: they are shared by GFCH and GFCH1, but for the sake of
% efficiency we duplicate this code in the modules calling them.

% research of the first bite of a word
first_bit(Ft, Value, Pos, BMask, DGNB, Res):-
  Index:= BMask /\ Value|
      first_bit1(Index, Ft, Value, Pos, BMask, DGNB, Res).

first_bit1(0, Ft, Value, Pos, BMask, DGNB, Res):-
  NValue := Value >> DGNB|
      first_bit(Ft, NValue, ~(Pos+1), BMask, DGNB, Res).
first_bit1(Index, Ft, _, Pos, _, DGNB, Res):-
  Index > 0,
  vector_element(Ft, Index, Loc)|
      Res:= Loc + (Pos * DGNB).

% research of the sum of the bites of a word
bit_sum(Dt, Value, Pos, BMask, DGNB, Res, Sum):-
  Pos > 0,
  NValue := Value >> DGNB,
  Index := BMask /\ Value,
  vector_element(Dt, Index, RLoc)|
      Sum1:= RLoc + Sum,
      bit_sum(Dt, NValue, ~(Pos-1), BMask, DGNB, Res, Sum1).
bit_sum(Dt, Value, 0, _, _, Res, Sum):-
  vector_element(Dt, Value, RLoc)|
      Res:=RLoc+Sum.
```

```
% searches for the best element to be propagated,
% it means the one having the smallest Domain Size DomSz,
% [DomSz, Dom |others]

prop_best([A,B|L], Min, C, Res, Couple):- A > Min|
      Res=[A,B|Res1],
      prop_best(L, Min, C, Res1, Couple).
prop_best([A,B|L], Min, C, Res, Couple):- A =< Min|
      Res=[Min, C|Res1],
      prop_best(L, A, B, Res1, Couple).
prop_best([], A, B, Res, Couple):- true|
      Res=[], [A, B]=Couple.


% the Lazy version is used in the core of the program
% searches for the best element to be propagated,
% it means the one having the smallest Domain Size DomSz,
% [DomSz, Dom |others]
% this lazy version stops as soon as it finds and element having
% a Domain size=1, [1, Dom, |others]. This works since we know that
% 0 elements cannot appear in the core of the program

lazy_prop_best(L, 1, C, Res, Couple):- true|
      Res=L, Couple=[1,C].
otherwise.
lazy_prop_best([A,B|L], Min, C, Res, Couple):- A > Min|
      Res=[A,B|Res1],
      lazy_prop_best1(L, Min, C, Res1, Couple).
lazy_prop_best([A,B|L], Min, C, Res, Couple):- A =< Min|
      Res=[Min, C|Res1],
      lazy_prop_best(L, A, B, Res1, Couple).
lazy_prop_best([], A, B, Res, Couple):- true|
      Res=[], [A, B]=Couple.

lazy_prop_best1([A,B|L], Min, C, Res, Couple):- A > Min|
      Res=[A,B|Res1],
      lazy_prop_best1(L, Min, C, Res1, Couple).
lazy_prop_best1([A,B|L], Min, C, Res, Couple):- A =< Min|
      Res=[Min, C|Res1],
      lazy_prop_best(L, A, B, Res1, Couple).
lazy_prop_best1([], A, B, Res, Couple):- true|
      Res=[], [A, B]=Couple.
```

3.5 FCH: utilities used by the problem solvers

```
%
%                          FORWARD CHECKING IN PARALLEL
%
%                     utilities used by SFCH, GFCH, SFCH1, GFCH1
%
%
%  Author:     Bernard Burg
%  Date:       5 july 90
%
%

:- module fch.
:- public single_bit_table/1,
          fch_input/4, write_in_file/2, gen_elements/6,
          sumlength/3, gen_decode/7, control_shoen/3, find_power/4,
          mpattern_table/6.



% utilites for SFCH and SFCH1
%--------------------------------
%
% single_bit_table(Table)
% The element number (2^N mod 37) of the table is:
% { 2^N, N }
    single_bit_table(Table) :- true|
new_vector(T0, 37),
sbt(31, T0, Table).

    sbt(0, T0, T) :- true| T = T0.
otherwise.
    sbt(N0, T0, T) :-
N1 := N0-1,
Shifted := 1 << N1,
Index := Shifted mod 37|
set_vector_element(T0, Index, _, {Shifted, N1}, T1),
sbt(N1, T1, T).


% utilities used by SFCH, SFCH1, GFCH, GFCH1
%-------------------------------------------

% checking the input parameters,
% ProNb has to be a power of 2
% 0 < DomSz =< 32

fch_input(DomSzIn, DomSzOut, ProNbIn, ProNbOut):-
  0 < DomSzIn, DomSzIn < 33|
    DomSzIn=DomSzOut,
    naive_bit_sum(ProNbIn, 0, BNB),
    (BNB =:= 1 -> ProNbIn=ProNbOut;
     otherwise;
     true ->
     util:p_console(
     string#"Error in the processors number ProNb. It  should be a power of 2", _)).
otherwise.
fch_input( _, _, _, _):- true|
```

```
      util:p_console(string#"Error in the domain size DomSz. (0 < DomSz =< 32)", _).

 %this is an elegant manner to count the number of bites in a word
 naive_bit_sum(0, Mem, Res):- true|
     Mem=Res.
 otherwise.
 naive_bit_sum(Nb, Mem, Res):-
   Nb1:= Nb - 1,
   Nb2:= Nb /\ Nb1|
     naive_bit_sum(Nb2, ~(Mem+1), Res).



 % utility, to write the result in a file
 %
 write_in_file(STR, File):- true|
       fel:open_file([print_length(32)|Stream], File, w, normal),
       write_in_file1(STR, Stream).

 write_in_file1([A|B], Stream):-
    vector_element(A, 1, A1),
    vector_element(A, 2, A2)|
       Resul=[A1, A2],
       Stream=[puttq(Resul)|Stream1],
       write_in_file1(B, Stream1).
 write_in_file1([], Stream):-true|
       Stream=[].

 % generates the domain elements, encoded in bites.
 % DomM is the whole domain
 % DomA is the domain with the even bites
 % DomB is the domain with the odd bites

 gen_elements(Val, ResM, ResA, ResB, ANb, BNb):- Val> 0|
       Loc:= 1 << Val,
       ResM:= Loc \/ ResM1,
       ResA:= Loc \/ ResA1,
       ANb:= ANb1 +1,
       gen_elements1(~(Val-1), ResM1, ResA1, ResB, ANb1, BNb).
 gen_elements(0, ResM, ResA, ResB, ANb, BNb):- true|
       ResM=1, ResB=0, ResA=1, ANb=1, BNb=0.

 gen_elements1(Val, ResM, ResA, ResB, ANb, BNb):- Val> 0|
       Loc:= 1 << Val,
       ResM:= Loc \/ ResM1,
       ResB:= Loc \/ ResB1,
       BNb:=BNb1 + 1,
       gen_elements(~(Val-1), ResM1, ResA, ResB1, ANb, BNb1).
 gen_elements1(0, ResM, ResA, ResB, ANb, BNb):- true|
       ResM=1, ResA=0, ResB=1, ANb=0, BNb=1.

 % the control of the shoen

 control_shoen([resource_low | Rpt], Ctl, Sync):-
     true|
         Ctl=[add_resource(1000000000),
 % for M-PSI  add_resource(1000000000, 1000000000),
             allow_resource_report | Ctl1],
         control_shoen(Rpt, Ctl1, Sync).
```

```
control_shoen([terminated | _], Ctl, Sync):-
    true|
        Sync=1, Ctl=[].
otherwise.
control_shoen([_|Y], Ctl, Sync):- true|
        control_shoen(Y, Ctl, Sync).

%  Compiles the list of constraints issued from the application
%  and gives the table of constraints used for the forward checking.
%  mpattern_table(L, MaxX, MaxY, T, Sync, CurPro)

mpattern_table(L, MaxX, MaxY, Tab, Sync, CurPro):- wait(Sync)|
      local_stats(SyncOut, CurPro),
      (wait(SyncOut) ->
        init_vector(ElemXY, MaxX, MaxY),
        new_vector(ElemY, MaxY),
        create_table(ElemY, ElemXY, 0, MaxY, ElemY1),
        new_vector(TabEmpt, MaxX),
        create_table(TabEmpt, ElemY1, 0, MaxX, TabInit),
        pattern_table(L, TabInit, Tab)).

% local_stats makes performances on the processor 1 only,
% to avoid unnecessary measures on other processors.
local_stats(SyncOut, 0):- true|
      util:req_time("Constraints", Top),
      (wait(Top) -> local_stats1(Top, SyncOut)).
otherwise.
local_stats(SyncOut, _):- true|SyncOut=1.
local_stats1(SyncIn, SyncOut):- wait(SyncIn)|
      util:req_red("red", SyncOut).

create_table(TabIn, Elem, Cur, Max, TabOut):- Cur < Max|
      set_vector_element(TabIn, Cur, _, Elem, TabIn1),
      create_table(TabIn1, Elem, ~(Cur+1), Max, TabOut).
create_table(TabIn, _, Max, Max, TabOut):- true|
      TabIn=TabOut.


pattern_table([A|B], TabIn, TabOut):- true|
      pattern_table1(A, TabIn, TabIn1),
      pattern_table(B, TabIn1, TabOut).
pattern_table([], TabIn, TabOut):- true|TabIn=TabOut.

pattern_table1([IndA, IndB|A], TabIn, TabOut):- true|
      pattern_table2(A, MaskIn, MaskOut),
      set_vector_element(TabIn, IndA, AV, AV1, TabOut),
      set_vector_element(AV, IndB, MaskIn, MaskOut, AV1).

pattern_table1([], TabIn, TabOut):- true| TabIn=TabOut.

pattern_table2([[Ind |A]|B], VMIn, VMOut):- true|
      pattern_mask(A, 0, NewMask),
      Mask:= NewMask /\ OldMask,
      set_vector_element(VMIn, Ind, OldMask, Mask, VMIn1),
      pattern_table2(B, VMIn1, VMOut).
pattern_table2([], VMIn, VMOut):- true|VMIn=VMOut.

% this allows to make an empty mask, without removing all the
% elements.
```

```
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).

init_vector(Vector, MaxX, MaxY):- true|
      new_vector(Vec, MaxX),
      Vect_value:= (1<<MaxY)-1,
      init_vector1(0, MaxX, Vect_value, Vec, Vector).

init_vector1(Ind, Max, Value, In, Out):- Ind < Max|
      set_vector_element(In, Ind, _, Value, In1),
      init_vector1(^(Ind+1), Max, Value, In1, Out).
init_vector1(Max, Max, _, In, Out):- true| In=Out.


% finds the power of a number

find_power(0, Val, In, Y):- true|
      Mask:=(In>>Val)/\1,
      find_power(Mask, ^(Val+1), In, Y).
find_power(1, Val, _, Y):- true| Y:=Val-1.


% utilities used by GFCH, GFCH1
%-----------------------------
% determines the number of words to use for decoding Y in
% check_pos 1 and 2
% we have to write it in a naive way since the compiler refuses
% the indexing with the modulo operation
sumlength(Size, Bitln, Res):-
  Test := Size mod Bitln|
    (Test =:= 0 ->  Res := (Size / Bitln) -1;
     otherwise;
     true ->        Res := (Size / Bitln)).

gen_decode(SizeX, SizeY, TabSZ, DT, FT, T_Size, MasTa):- true|
table_size(SizeY, TabSZ, T_Size),
        word_mask(^(T_Size-1), MasTa),
        digit_table(T_Size, DT),
        FTln:= 1<<T_Size,
        new_vector(FTIn, FTln),
        first_table(^(FTln-1), FTIn, FT).

% calculation of the table length
table_size(Size, Prop, Res):-
  Test:= Size mod Prop|
   (Test =:= 0 ->  Res:= Size / Prop;
    otherwise;
    true ->        Res:= (Size / Prop)+1).

word_mask(Len, Bits):- Len > -1|
      Bits:= Bits1 \/ (1 << Len),
      word_mask(^(Len-1), Bits1).
word_mask(-1, Bits):- true|Bits:=0.
```

```
% generation of the table recording the number of bites of
% a word

digit_table(Max, Tr):-
  Size:= 1<<Max|
      new_vector(T, Size),
      set_vector_element(T, 1, _, 1, T1),
      digit_table1(2, ~(Max+1), T1, Tr).
digit_table1(N, Max, T1, Tr):-
  N < Max,
  Locmin:= 1 << (N-1),
  Locmax:= 1 << N,
  Mask:= Locmin xor (-1)|
      digit_table2(Mask, Locmin, Locmax, T1, T2),
      digit_table1(~(N+1), Max, T2, Tr).
digit_table1(Max, Max, T, Tr):- true|T=Tr.

digit_table2(Mask, N, Locmax, T1, Tr):-
  N < Locmax,
  Ind:= N /\ Mask,
  vector_element(T1, Ind, Value)|
      set_vector_element(T1, N, _, ~(Value+1), T2),
      digit_table2(Mask, ~(N+1), Locmax, T2, Tr).
digit_table2( _, Max, Max, T, Tr):- true|T=Tr.


% generation of the table recording the first bite of the word.
% this algorithm could also take the same principle as the
% digit_table algorithm (but we kept a more classical way of doing).

first_table(Max, Yvalin, Yvalout):- Max > 0|
      find_power(0, 0, Max, Y),
      (wait(Y) ->
      set_vector_element(Yvalin, Max, _, Y, Yvalin1),
      first_table(~(Max-1), Yvalin1, Yvalout)).
first_table(0, Yvalin, Yvalout):- true|Yvalin=Yvalout.
```

Bibliography

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

[2] 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.

[3] 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.

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