HAL
open science

Automatic parallelization of structured if statements without if conversion

M.C. Giboulot, François Thomasset

To cite this version:

M.C. Giboulot, François Thomasset. Automatic parallelization of structured if statements without if conversion. [Research Report] RR-1408, INRIA. 1991. inria-00075152

HAL Id: inria-00075152

https://inria.hal.science/inria-00075152

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél (1) 39 63 55 11

Rapports de Recherche

N° 1408

Programme 2
Calcul Symbolique, Programmation
et Génie logiciel

AUTOMATIC PARALLELIZATION OF STRUCTURED IF STATEMENTS WITHOUT IF CONVERSION

Marie-Christine GIBOULOT
François THOMASSET

Avril 1991

RR_1408

Automatic parallelization of structured IF statements without IF conversion

Parallélisation automatique des instructions IF structurées sans génération de masques

Marie Christine GIBOULOT
GIPSI S.A.
2, Boulevard Vauban, 78180 Montigny le Bretonneux

and

François THOMASSET
INRIA - Rocquencourt
BP 105 - 78153 Le Chesnay cedex

June 6, 1991

Abstract

We propose an algorithm for the treatment of structured IF statements in automatic parallelization of FORTRAN-77 programs; compared with previous algorithms this does not require the conversion of the IF statements to 'masked assignments'. Control dependences are added to the graph, and classical loop distribution is applied until a control dependence crosses a loop boundary; the IF instructions is then broken in two new IF instructions. The algorithm is implemented in the automatic parallelizer available under PIAF, an Interactive Programming environment for FORTRAN. This work was performed within the framework of ESPRIT project 2177 GIPE-II.
Keywords: parallelization, FORTRAN, programming environment, CENTAUR

Résumé

Cet article décrit un algorithme de parallélisation automatique de FORTRAN-77, qui traite les instructions IF structurées sans générer d'instructions 'masquées'. On introduit des dépendances de type 'contrôle' dans le graphe, et on applique l'algorithme classique de distribution de boucle; lorsqu'une dépendance de contrôle est amenée à traverser une frontière de boucle, l'instruction IF correspondante est décomposée en deux nouvelles instructions IF. L'algorithme est implémenté dans le paralléliseur automatique disponible sous PIAF, environnement de Programmation Interactif pour FORTRAN. Ce travail a été réalisé dans le cadre du projet ESPRIT 2177 GIPE-II.
Mots Clés: parallelisation, FORTRAN, environnement de programmation, CENTAUR
This paper is submitted to 2nd Symposium on High Performance Computing.

1 Classical view of loop distribution

Our purpose is to improve parallelization of loops containing structured IF statements, within the framework of an interactive parallelizer; we try to keep close to the original structure given by the user.
We review the well known algorithm of loop distribution ([11],[15],[16]) for simply nested loops:

- computation of the dependence graph;
- computation of the reduced graph using Tarjan's algorithm [2];
- topological sort of reduced graph;
- for each strongly connected component, emission of either a sequential loop or a parallel loop;
- whenever possible merge loops of similar structure. In fact merging can be performed simultaneously with loop emission: we gather the instructions which can be in one loop into *regions*, and emit a loop only when a region is complete.

In order to deal with IF statements one method (IF-conversion) consists in converting the loop body to a sequence of so called masked instructions [1], and then apply the classical algorithm; the mask of an instruction is a boolean expression guarding the execution of each instruction; this is illustrated by the example of figure 1.

```
C EXAMPLE 1: NESTED IF STATEMENTS
      DO I = 1, 100
S1::     IF (X(I).GT.0) THEN
S2::       IF (Y(I).LE.1) THEN
S3::          A(I) = B(I)+C(I)
              ELSE
S4::          D(I) = A(I) + 1
           ENDIF
         ELSE
S5::       IF (Y(I)+X(I).GT. 30) THEN
S6::          W(I+1) = W(I) +B(I)-C(I)+ A(I-1)
              ELSE
S7::          D(I) = W(I) + A(I-1) - 1
           ENDIF
         ENDIF
      END DO
```

Figure 1: First example

The mask for the 'then' branch is `(X(I).GT.0)` while that of the 'else' branch is the complement; thus the masked instructions are as shown on figure 2. From this the standard distribution [1] algorithm may yield the loop on figure 3.

[1] Dependences are computed by adding the parameters of the mask to the input of the statement.

```
DO I = 1, 100
 IF (X(I).GT.0 .AND. Y(I).LE.1) A(I) = B(I)+C(I)
 IF (X(I).GT.0 .AND. .NOT. Y(I).LE.1) D(I) = A(I) + 1
 IF (.NOT. X(I).GT.0 .AND. Y(I)+X(I).GT. 30)
.    W(I+1) = W(I) +B(I)-C(I)+ A(I-1)
 IF (.NOT. X(I).GT.0 .AND. .NOT. Y(I)+X(I).GT. 30)
.    D(I) = W(I) + A(I-1) - 1
END DO
```

Figure 2: Example 1: result of IF conversion

```
DOALL I = 1, 100
 IF (X(I).GT.0 .AND. Y(I).LE.1) A(I) = B(I)+C(I)
 IF (X(I).GT.0 .AND. .NOT. Y(I).LE.1) D(I) = A(I) + 1
END DOALL
DO I = 1, 100
 IF (.NOT. X(I).GT.0 .AND. Y(I)+X(I).GT. 30)
.    W(I+1) = W(I)+B(I)-C(I)+A(I-1)
END DO
DOALL I = 1, 100
 IF (.NOT. X(I).GT.0 .AND. .NOT. Y(I)+X(I).GT. 30)
.    D(I) = W(I) + A(I-1) - 1
END DOALL
```

Figure 3: Example 1: loop distribution after IF conversion

We note several drawbacks to this approach:

- A significant increase of computations and memory references to build the boolean conditions; this is all the more important as the masks must also be computed in the sequential sections. Of course recomputation of boolean conditions may be avoided by saving them in auxiliary variables, but systematic use of temporary variables increases memory usage, or it may incur overhead due to loop splitting (note however that the present technique may also need to save the conditions in auxiliary variables in some conditions in order to preserve semantics).
- A simplifier for boolean expressions may be required (for the implementation of boolean expression simplifiers in a vectorizer see [12]).
- The structure of the program is lost which makes it difficult to explain the transformation in an interactive environment, especially in case of nested IF statements.

2 Principles of present algorithm

First we state that our computational model in this paper is a multiprocessor as described in the EWS project [13] [2]:

- The parallel loops are DOALL loops, i.e. different iterations are totally independent and do not require synchronization.
- For simplicity we consider simply nested loops, with assignments and structured IF statements (IF-THEN-ELSE) [3]

Our purpose is to keep the structure of the loop body unchanged if possible; in this perspective we define the dependence graph as follows:

- Set of nodes:
  - to each assignment statement one node is associated;
  - to the boolean condition of each IF statement one node is associated.
- The edges are:
  - data dependence edges (true, anti, output, noted respectively $\longrightarrow$, $\dashrightarrow$, $\overset{o}{\dashrightarrow}$)
  - control dependence edges (noted $\overset{c}{\dashrightarrow}$) from the IF condition to the instructions in the TRUE and FALSE branches.

Thus in this approach control dependences are defined directly from the abstract syntax (as opposed the PDG approach, [4], [7]); this is summed up by the following rules, with N(A) being the node associated to an instruction:

A ≡ IF cond
THEN { B1, B2,... }
ELSE { C1, C2,... }
N(A) $\overset{c}{\dashrightarrow}$ N(B1), N(A) $\overset{c}{\dashrightarrow}$ N(B2),... ,N(A) $\overset{c}{\dashrightarrow}$ N(C1)...

In the rest of the paper we identify A and N(A).

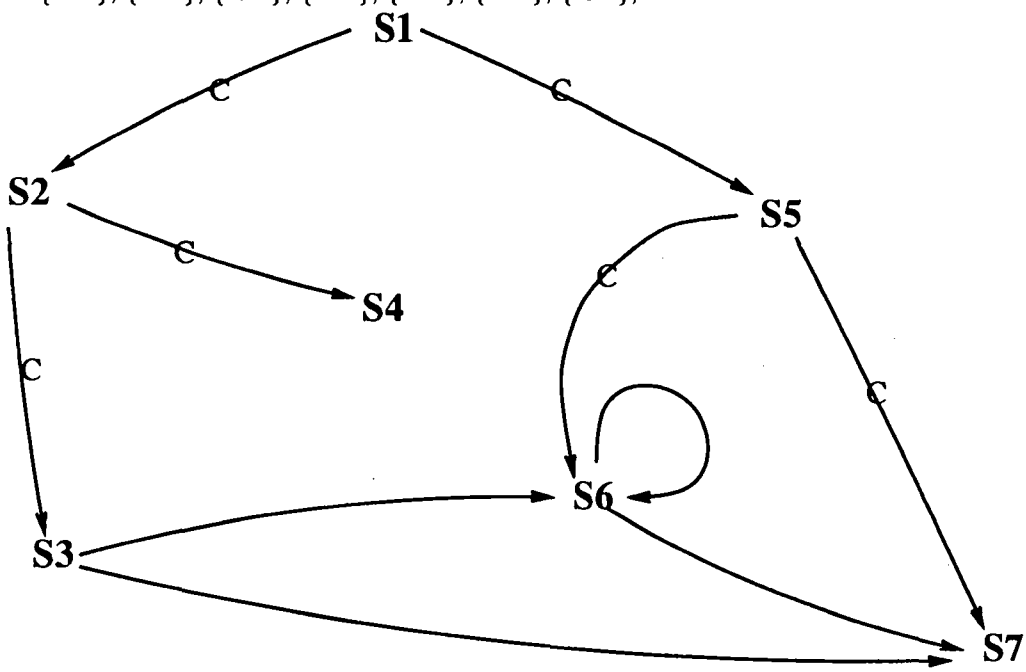
We note δ an unspecified *data* dependence:

$\delta \equiv \longrightarrow \cup \dashrightarrow \cup \overset{o}{\dashrightarrow}$

[2] ESPRIT project, Euro Work Station project 2569

[3] the application to multiply nested loops will be considered in a forthcoming paper

In example of figure 1 we have the following graph:
S1 $\xrightarrow{c}$ S2 S1 $\xrightarrow{c}$ S5,
S2 $\xrightarrow{c}$ S3, S2 $\xrightarrow{c}$ S4
S5 $\xrightarrow{c}$ S6, S5 $\xrightarrow{c}$ S7
S3 $\longrightarrow$ S6, S6 $\longrightarrow$ S6, S3 $\longrightarrow$ S7, S6 $\longrightarrow$ S7
In this graph there are 7 strongly connected components:
{ S1 }, { S2 }, { S3 }, { S4 }, { S5 }, { S6 }, { S7 };



S6 corresponds to a recurrence and therefore should be in a sequential loop; however the other components can be executed in parallel; furthermore S6 has to wait for completion of S3; therefore we have to split the original IF statement S1 as illustrated below:

```
C$PARALLEL_SECTION
C$DO_PARALLEL 12001
C$LOCAL i
      do 12001 i=1,100,1
        if (x(i).gt.0) then
          if (y(i).le.1) then
            a(i) = b(i)+c(i)
          else
            d(i) = a(i)+1
          endif
        endif
12001   continue
C$DO_SEQUENTIAL 12002,AFTER(12001)
C$LOCAL i
      do 12002 i=1,100,1
        if (.not.x(i).gt.0) then
          if (y(i)+x(i).gt.30) then
            w(i+1) = (w(i)+b(i))-c(i)+a(i-1)
          endif
        endif
```

```
12002   continue
C$DO_PARALLEL 12003,AFTER(12002)
C$LOCAL i
      do 12003 i=1,100,1
        if (.not.x(i).gt.0) then
          if (.not.y(i)+x(i).gt.30) then
            d(i) = (w(i)+a(i-1))-1
          endif
        endif
12003   continue
C$END_PARALLEL_SECTION
```

Two points must be stressed here:

1. If the parameters of the condition are modified, it may become necessary to save the condition into a temporary (examples of this will be shown further below);

2. As mentioned above we do not want to introduce synchronizations between iterations of a parallel loop. Therefore a dependence of non-zero distance between two instructions forces these instructions to be in different parallel loops; this is illustrated by the example of figure 4.

```
C EXAMPLE 2:  ORIGINAL LOOP
      DO 1 I = 1, 100
        IF (X(I).GT.0) THEN
           A(I) = B(I)+C(I)
         ELSE
           D(I) = A(I-1)+B(I)-C(I)
        ENDIF
1     CONTINUE
```

```
C EXAMPLE 2:  TRANSFORMED LOOP
C$PARALLEL_SECTION
C$DO_PARALLEL 12001
C$LOCAL i
      DO 12001 I=1,100,1
         IF (X(I).GT.0) THEN
           A(I) = B(I)+C(I)
         ENDIF
12001 CONTINUE
C$DO_PARALLEL 12002,AFTER(12001)
C$LOCAL I
      DO 12002 I=1,100,1
         IF (.NOT.X(I).GT.0) THEN
           D(I) = (A(I-1)+B(I))-C(I)
         ENDIF
12002 CONTINUE
C$END_PARALLEL_SECTION
```

Figure 4: Example 2

This leads to the following definitions:

- We abbreviate *component* for strongly connected component of the dependence graph.
- A component is *parallel* if it does not contain a cycle, otherwise it is *sequential* (note that a parallel component may contain only one node).
- Two instructions are *compatible* if there is no loop-carried dependence between them (i.e. dependence of non zero distance). Our goal is to put compatible components into the same loop.
  We note : 'A # B' if A and B are non compatible instructions, and we extend the # relation to components.

3 The basic algorithm

The main loop of the algorithm is shown in figure 8: this is derived from the distribution algorithm in [3]. We visit the graph in topological order: for this purpose, the algorithm successively emits new loops containing the image of the nodes, of type either sequential or parallel. Of course the nodes which belong to one strongly connected component must be emitted all at once. We call a *region* a list of mutually compatible strongly connected components. The region which will define the body of current loop is progressively constructed in variable *R*. We maintain a list (variable *nopreds*) of strongly connected components which are:

- either without predecessors,
- or whose predecessors have already been emitted in a new loop.

At any step of the algorithm, a component C_S , member of *nopreds* is a candidate for joining the region R and defining the current loop; however in order to validly join the region R, C_S must be *compatible* with *all* other members of R: variable *nonOK* collects those components which are temporarily non compatible with R. In case we find such a component C_S we visit its successors (procedure *visit-sons*, see figure 9) in order to spot any other candidates for joining the current region. In this way regions are constructed which contain compatible components.

```
        DO 100 I = 1,100
A::        Y(I) = 3 * I
B::        IF (X(I).GT.0) THEN
C::            Y(I) = -Y(I)
           END IF
D::        Z(I) = Y(I) + U(I)
100     CONTINUE
```

Figure 5: Example 3

Meaning of Control Dependence: If B $\xrightarrow{e}$ C (example 3), then C must not be issued before B; however in this example, the topological sort might give either {A, B, C, D } (i.e. the original order), or {B, A, C, D }; therefore we need to enforce the order given in the original program, which may be defined by the following rules:

$$\frac{\text{L is a list of instructions, } L = A :: B :: L'}{A \ll B}$$

$$\frac{\text{L is a list of instructions, } A \ll B,\ B \ll C}{B \ll C}$$

$$\frac{A \ll B \text{ in C.THEN-BLOCK, } C \text{ is an IF statement}}{A \ll B}$$

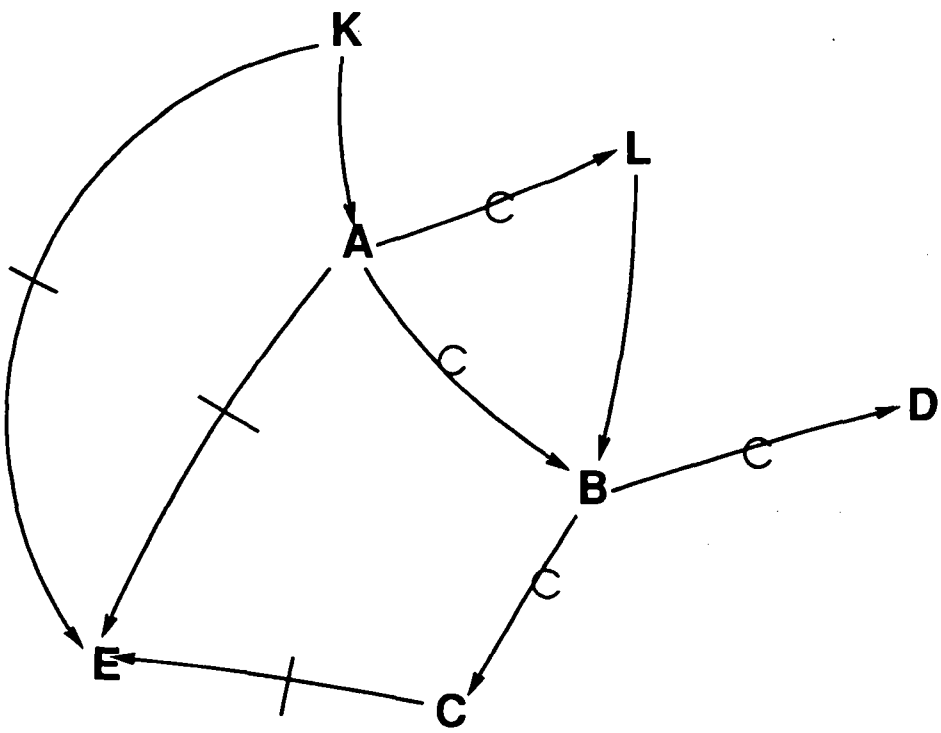
$$\frac{A \ll B \text{ in C.ELSE-BLOCK, } C \text{ is an IF statement}}{A \ll B}$$

```
C EXAMPLE 4 :
      DO I =1, N
A::     IF (X(I).GT.0) THEN
B::       IF (X(I).LT.200) THEN
C::        A(I) = 3
           ELSE
D::        B(I) = C(I) - 1
          ENDIF
        ENDIF
E::     X(I-1) = A(I+1)
      ENDDO
```

```
C EXAMPLE 4:  LOOP WITH MASKS
      DO I =1, N
K::     MASK1(I) = (X(I).GT.0)
A::     IF (MASK1(I)) THEN
L::       MASK2(I) = (X(I).LT.200)
B::       IF (MASK2(I)) THEN
C::        A(I) = 3
           ELSE
D::        B(I) = C(I) - 1
          ENDIF
        ENDIF
E::     X(I-1) = A(I+1)
      ENDDO
```

Figure 6: Example 4

Next consider example 4 (figure 6): the boolean conditions are saved in temporaries; the graph is shown below, and is seen to be acyclic:



Now because of the definition of the # relation, we get the following regions (the anti dependences have distance 1):

{ K , A , B , L , D } , { E } , { C }

This will lead to break the IF statement A as shown on figure 7

```
C EXAMPLE 4:  PARALLELIZED LOOP
C$PARALLEL_SECTION
C$DO_PARALLEL 12001
C$LOCAL I
      DO 12001 I =1, N
K::     MASK1(I) = (X(I).GT.0)
A::     IF (MASK1(I)) THEN
L::       MASK2(I) = (X(I).LT.200)
B::       IF (.NOT. MASK2(I)) THEN
D::        B(I) = C(I) - 1
          ENDIF
        ENDIF
12001 CONTINUE
C$DO_PARALLEL 12002,AFTER(12001)
C$LOCAL I
      DO 12002 I =1, N
E::     X(I-1) = A(I+1)
12002 CONTINUE
C$DO_PARALLEL 12003,AFTER(12002)
C$LOCAL I
      DO 12003 I =1, N
A'::    IF (MASK1(I) .AND. MASK2(I)) THEN
C ::       A(I) = 3
        ENDIF
12003 CONTINUE
```

Figure 7: Example 4

Remark: Influence of the definition of #: Now if we had another target machine, we might want to modify the definition of #: for instance for a vector architecture, we would like to say that two instructions are compatible even if they have dependences with non zero distance (i.e. loop carried dependences); but with such a definition, we would run into difficulties: we would find in example 4 that all components are compatible; in this case the original order cannot help since E has to be executed before C; furthermore we would have to interrupt the emission of B in order to generate E. Since we project to generate vector FORTRAN for a SIMD machine, we plan to solve this difficulty by imposing the same constraints on # for SIMD as for MIMD machines (with the exception of antipendences of instructions on themselves).

Splitting IF statements: However it may happen that the component of the node corresponding to the boolean condition of an IF statement comes into the current region, and some control dependent node comes into a different region (see for instance example of figure 4); in order to correct this situation procedure *look-at-if* examines (figure 10), in a region which has just been completed, all "IF" nodes A, such that the component of A is in R; for such an A procedure process-if is invoked, and A is split, if necessary, into two new IF statements, so that ultimately no control dependence crosses a region boundary.

DISTRIB ALGORITHM:

INPUT:

dependence graph of a DO loop

OUTPUT:

list-R: sequence of *regions*; each region is a list of compatible components to be embedded in one loop.

NOTATION:

```
preds(C) = set of components Cs such that Cs δ C
BEGIN
list-R ← {}
nopreds ← {Cs such that : preds(Cs ) = ∅}
sort nopreds with respect to relation "≪ " (see page 7)
already-seen ← ∅
WHILE nopreds ≠ ∅
 nonOK ← ∅
 R ← ∅
 currenttype ← nil
 /* currenttype: type of current region: parallel or sequential */
 WHILE nopreds \ nonOK ≠ ∅
    IF currenttype=nil
      THEN
       Cs ← first element of nopreds \ nonOK;
       currenttype ← type(Cs )
      ELSE
       Cs ← an element of nopreds \ nonOK
        such that type(Cs )=currenttype
       IF such an element does not exist
        THEN Cs ← nil; nonOK ← nopreds
       ENDIF
    ENDIF
    IF Cs ≠ nil
      THEN
       nopreds ← nopreds \ {Cs }
       already-seen ← already-seen ∪ {Cs }
       R ← R :: {Cs }
       (nopreds, nonOK) ← visit-sons (Cs , nopreds, nonOK)
    END IF
 END WHILE
 (R,G, already-seen, nopreds) ← look-at-if (R,G,already-seen, nopreds)
 list-R ← list-R :: R
END WHILE
```

END DISTRIB

Figure 8: distribution algorithm

```
visit-sons (C, nopreds, nonOK)
INPUT : a component C,
        the set already-seen
OUTPUT: nonOK and nopreds are modified
BEGIN
  FOR EACH Cr such that Cδ Cr
    IF (C# Cr)
      THEN nonOK ← nonOK ∪ {Cr}
    END IF
    IF preds(Cr) ⊆ already-seen
      THEN nopreds ← nopreds ∪ {Cr}
    ENDIF
  END FOR EACH
  RETURN (nopreds, nonOK)
END visit-sons
```

Figure 9: visit-sons procedure

```
look-at-if (R,G,already-seen, nopreds)
INPUT: R is a list of compatible components
       dependence graph G
OUTPUT: R, G, already-seen, nopreds
BEGIN
  insts-R ← set of instructions contained in the components of R:
    /* insts-R = ⋃ { A such that comp(A) ∈ R } */
  todo ← insts-R
  WHILE todo ≠ ∅
    A ← an element of todo
    todo ← todo \ { A }
    IF A is an IF instruction
      THEN
        (A1,A2,A3,R,G,already-seen,nopreds) ← process-if(R,G,A,already-seen,nopreds)
    ENDIF
  END WHILE
  /* NOW NO CONTROL DEPENDENCE CROSSES BOUNDARY OF R */
  RETURN (R,G, already-seen, nopreds)
END look-at-if
```

Figure 10: look-at-if procedure

procedure process-if: This procedure examines an IF statement A, and returns 3 new statements:

- A1 saves the condition into a temporary;
- A2 is an IF statement compatible with the components already in R;
- A3 is an IF statement non compatible with R;
- it may happen that some of these statements are empty.
- Whenever A contains a nested IF statement, process-if is called recursively.

Saving Boolean Conditions: We need to save the condition in a temporary when some of its variables are modified during execution of the loop, i.e. there is some S satisfying: $A \dashv\!\vdash S$; [4] we distinguish two cases:

- save condition, first case: The variables of the condition are modified by an instruction S such that:
  - $A \dashv\!\vdash S$,
  - and S compatible with A (comp(S) $\in$ R),
  - yet the IF statement must be split (some instruction B, which is control dependent on A, is incompatible with A).

This is the case in the example shown of figure 11.

```
C Example 5:  ORIGINAL LOOP
      DO 1 I = 2, 200
A::      IF (C(I).NE.C(I-1)) THEN
B::        A(I+1) = B(I) - 5
         ELSE
S::        C(I) = B(I) + B(I+1)
         ENDIF
1     CONTINUE
```

```
C Example 5:  TRANSFORMED LOOP
      logical mask(200)
C$PARALLEL_SECTION
C$DO_SEQUENTIAL 12001
C$LOCAL i
      do 12001 i=2,199,1
         mask(i) = c(i).ne.c(i-1)
         if (.not.mask(i)) then
            c(i) = b(i)+b(i+1)
         endif
12001 continue
C$DO_PARALLEL 12002,AFTER(12001)
C$LOCAL i
      do 12002 i=2,199,1
         if (mask(i)) then
            a(i+1) = b(i)-5
         endif
12002 continue
C$END_PARALLEL_SECTION
```

Figure 11: Example 5: $A \dashv\!\vdash S$

[4] we assume the absence of side effects, so that computation of the condition cannot modify any variable; also if there was some dependence from S to A, then A and S would belong to a cycle, because of the control dependence from A to S, and sequential execution always preserves dependences.

• save condition, second case: The variables of the condition are modified by an instruction P such that:
– $A \not\rightarrow P$,
– the component of P is parallel,
– but the antidependence has a non zero distance, so that A and P are incompatible: see example on figure12.

```
C Example 6:  ORIGINAL LOOP
      DO 1 I = 2, 200
A::      IF (A(I).GT.0) THEN
P::         A(I-1) = B(I) + 4
          ELSE
B::         C(I) = B(I)*2
         ENDIF
1     CONTINUE
```

```
C Example 6:  TRANSFORMED LOOP
      logical mask(200)
C$PARALLEL_SECTION
C$DO_SEQUENTIAL 12001
C$LOCAL i
C$END_PARALLEL_SECTION
      do 12001 i=2,200,1
         if (.not.a(i).gt.0) then
            c(i) = b(i)*2
         endif
12001 continue
C$DO_PARALLEL 12002
C$LOCAL i
      do 12002 i=2,200,1
         mask(i) = a(i).gt.0
12002 continue
C$DO_PARALLEL 12003,AFTER(12002)
C$LOCAL i
      do 12003 i=2,200,1
         if (mask(i)) then
            a(i-1) = b(i)+4
         endif
12003 continue
```

Figure 12: Example 6: $A \not\rightarrow P$

Remark: We assume that there is no side effect in evaluation of boolean conditions; let A be an IF statement A with parallel type (the type of comp(A) is parallel), and B a control dependent statement on A (or such that there is a path of control dependences from A to B):

A $\xrightarrow{c}$ * B

then the only possible dependence from A to B is A $\dashrightarrow$ B.'(otherwise there would exist a cycle containing A); we use this property to update the graph.

<u>**process-if**</u> : proc (R, G, A, already-seen, nopreds)

<u>INPUT :</u>

R = ordered list of compatible components
(candidates for embedding into one loop)
G = original graph
A = an IF instruction; at this point comp(A) $\in$ R
/* so that comp(A) is compatible with R */

<u>OUTPUT:</u>

A1, A2, A3 = sequence of instructions replacing A:
A1 is an assignment instruction saving the condition of A in a temporary
A2, A3 are IF statements
(some of A1, A2, A3 may be nil)
R' = ordered list of compatible components,
such that:
if A1 $\neq$ nil then comp(A1) $\in$ R'
if A2 $\neq$ nil then comp(A2) $\in$ R'
if comp(A2) $\in$ R' and A2 $\xrightarrow{c}$ B then comp(B) $\in$ R'
(control dependences must not cross loop boundaries)
G' = updated graph

<u>MODIFIED:</u>

already-seen
nopreds: nodes without predecessor

Figure 13: process-if parameters

```
BEGIN process-if
 /* INITIALIZATIONS */
 A1 ← nil; A2 ← nil; A3 ← nil
 BT ← nil; BF ← nil; BT' ← nil; BF' ← nil
 /* BT and BF are intended to contain the blocks for TRUE and FALSE branches of A2 */
 /* BT' and BF' are likewise TRUE and FALSE blocks for A3 */
 R' ← R
 G' ← G
 /* COMPUTE BLOCK IN THE TRUE BRANCHES */
 (BT, BT',R',G', already-seen, nopreds) ← compute-blocks (blocktrue(A),R',G', already-seen, nopreds)
 /* SAME FOR BLOCK FALSE */
 (BF, BF',R',G', already-seen, nopreds) ← compute-blocks (blockfalse(A),R',G', already-seen, nopreds)
 IF BT ≠ nil OR BF ≠ nil
  THEN
    IF BT' =nil AND BF'=nil
      THEN
        RETURN (A1=nil, A2=A, A3=nil, R'=R, G'=G)
    ELSE
        A2 :: "if cond(A) then BT else BF endif"
  ENDIF
 ENDIF
 A3 :: "if cond(A) then BT' else BF' endif"
  /* REMARKS: A1, A2 may be nil; A3 is never nil */
  /* cond(A) may need to be saved*/
 IF (∃ S such that comp(S) ∈ R' AND A ⊣→ S in G)
    OR (∃ P such that comp(P) is parallel
      AND A ⊣→ P in G
      AND the distance of this dependance is not 0)
  THEN
  /* THE BOOLEAN CONDITION OF A IS MODIFIED */
    (A1, A2, A3) ← save-cond (A, A1, A2, A3)
 ENDIF
 /* UPDATE DEPENDENCES: */
 G' ← update-dep (A1,A2,A3,A,G')
 /* FIND COMPONENTS OF A1, A2, A3 */
 R' ← find-components (A1,A2,A3,A,G',R')
 remove A from G'
 already-seen ← already-seen ∪ { comp(A1), comp(A2) } \ { comp(A) }
 nopreds ← nopreds ∪ { comp(A3) } \ { comp(E) such that A3 δ E }
 RETURN (A1, A2, A3, R', G', already-seen, nopreds)
end process-if
```

Figure 14: process-if procedure code

```
compute-blocks: proc (a-block, R', G', already-seen, nopreds)
  INPUT: a-block
  OUTPUT: new-block: contains statements compatible with R'
    new-block' : statements non compatible with R'
  MODIFIED: R', G', already-seen, nopreds
BEGIN
  FOR B in a-block
    IF B is an 'assign' stmt
      THEN
        IF comp(B) ∈ R'
          THEN new-block ← new-block :: B
          ELSE new-block' ← new-block' :: B
        ENDIF
    ENDIF
    IF B is an 'if' stmt
      THEN
        IF comp(B) ∈ R'
          THEN
            (B1, B2, B3, R', G', already-seen, nopreds) ← process-if (R', G', B, already-seen, nopreds)
            new-block ← new-block :: B1 :: B2
            new-block' ← new-block' :: B3
          ELSE new-block' ← new-block' :: B
    ENDIF
  END FOR
  RETURN (new-block, new-block',R',G', already-seen, nopreds)
end compute-blocks
```

Figure 15: Construction of new blocks

```
save-cond: proc (A, A1, A2, A3)
BEGIN
 TEMP : a new array variable
 A1 :: TEMP(indx) = cond(A)
 replace cond(A2) and cond(A3) by TEMP(indx)
 RETURN (A1, A2, A3)
end save-cond
```

Figure 16: Save Condition Procedure

```
update-dep : proc (A1,A2,A3,A,G')
  INPUT: A1, A2, A3: new instructions to replace A
    A: an old IF statement
  OUTPUT: new graph G'
BEGIN
  /* UPDATE DEPENDENCES: */
  /* make instructions in BT, BF control dependent on A2 in G' */
  /* make instructions in BT', BF' control dependent on A3 in G' */
  /* note: A1, A2 may be nil, but A3 ≠ nil */
  IF A1 ≠ nil THEN add A1 to nodes of G' ENDIF
  IF A2 ≠ nil THEN add A2 to nodes of G' ENDIF
  add A3 to nodes of G'
  IF A1 ≠ nil and A2 ≠ nil THEN A1 ⟶ A2 in G' ENDIF
  IF A1 ≠ nil THEN A1 ⟶ A3 in G' ENDIF
  FOR ALL B such that B -e→ A in G'
    { IF A1 ≠ nil THEN B -e→ A1 in G' ENDIF
      IF A2 ≠ nil THEN B -e→ A2 in G' ENDIF
      B -e→ A3 in G' }
  END FOR
  FOR ALL B such that B ⟶ A in G'
    { IF A1 = nil
      THEN
        IF A2 ≠ nil THEN B ⟶ A2 in G' ENDIF
        B ⟶ A3 in G'
      ELSE
        B ⟶ A1 in G'
    }
  END FOR
  FOR ALL B such that A -+→ B in G
    { IF A1 = nil
      THEN
        IF A2 ≠ nil THEN A2 -+→ B in G' ENDIF
        A3 -+→ B in G'
      ELSE
        A1 -+→ B in G'
    }
  END FOR
  RETURN G'
end update-dep
```

Figure 17: Update Dependence Procedure

```
find-components: proc (A1,A2,A3,A,G',R')
  INPUT: A1, A2, A3: new instructions to replace A
    A: an old IF statement
    G': current dependence graph
  OUTPUT: R'
    components of A1, A2, A3 are updated
BEGIN
  CASE
  A1 ≠ nil
    ⟹
      IF A is sequential
        THEN
          G" ← subgraph (G', nodes(comp(A)) \ {A } ∪ { A1, A2 }
          comp(A1) ← G" (sequential)
          comp(A2) ← G" (sequential)
          replace in R' comp(A) by G"
        ELSE
          comp(A1) ← { A1 } (parallel)
          comp(A2) ← { A2 } (parallel)
          replace in R' comp(A) by comp(A1) followed by comp(A2)
      ENDIF
  A1=nil, A2 ≠ nil
    ⟹
      G" ← subgraph (G', nodes(comp(A)) \ {A } ∪ { A2 }
      comp(A2) ← G"
      replace in R' comp(A) by G"
  A1=nil, A2 = nil
    ⟹
      remove comp(A) from R'
  END CASE
  comp(A3) ← { A3 }
  type(comp(A3)) is type of the first component to study
  /* in the sense defined by the topological sort
    of the reduced graph of G' */
  RETURN R'
end find-components
```

Figure 18: Update Components Procedure

4 A Case Study

Let us give the steps of the algorithm for example on figure 1; as mentioned above there are 7 components:
{ S1 }, { S2 }, { S3 }, { S4 }, { S5 }, { S6 }, { S7 };

1. *build the first region:*

We select C_S =comp(S1), then add to R comp(S2), comp(S5) by calling visit-sons(comp(S1)); another call to visit-sons on comp(S5) does not allow to increase R for non compatibility reasons. We therefore get the first region:

R1= { comp(S1), comp(S2), comp(S3), comp(S4), comp(S5) }

2. look-at-if(R1):

```
  process-if(comp(S1)):
    process-if(comp(S2)):
      BT = S3, BT' = nil
      BF = S4, BF' = nil
      A1 = nil, A2=S2 , A3=nil
    return from process-if(comp(S2))
    process-if(comp(S5)):
      BT = nil, BT' = S6
      BF = nil, BF' = S7
      A1 = nil, A2=nil
      A3 = S5' = IF (Y(I)+X(I).GT. 30) THEN S6 ELSE S7
    return from process-if(comp(S5))
    BT = S2, BT'=nil
    BF = nil, BF' = S5'
      with S5' = IF (Y(I)+X(I).GT. 30) THEN S6 ELSE S7
    A1 = nil
    A2 = S1' = IF (X(I).GT.0) THEN S2
    A3 = S1" = IF (.NOT.X(I).GT.0) THEN S5'
  return from process-if(comp(S1))
  modified region : R1'= { comp(S1'),comp(S2),comp(S3),comp(S4) }
  return from look-at-if(R1)
```

3. *look-at-if(R2)*

```
  with R2 = { comp(S1"), comp(S5') }
  process-if(comp(S1"))
    process-if(comp(S5'))
      BT = S6, BT' = nil
      BF = nil, BF' = S7
      A1 = nil
      A2 = S5" = IF (Y(I)+X(I).GT. 30) THEN S6
      A3 = S5"' = IF (.NOT. Y(I)+X(I).GT. 30) THEN S7
    return from process-if(comp(S5'))
    BT = S5" , BT' = S5"'
    BF = nil, BF' = nil
    A1 = nil
    A2 = S1"' = IF (.NOT.X(I).GT.0) THEN S5"
    A3 = S1"" = IF (.NOT.X(I).GT.0) THEN S5"'
  return from process-if(comp(S1"))
  modified region : R2' = { comp(S1"'), comp(S5"), comp(S6) }
```

4. look-at-if (R3)

```
  with R3 = { comp(S1""), comp(S5"'), comp(S7) }
  process-if(comp(S1""))
    process-if(comp(S5"'))
      BT = S7 , BT' = BF = BF' = nil
```

A1 = nil, A2 = S5"', A3 = nil
return from process-if(comp(S5"'))
BT = S5"', BT' = BF = BF' = nil
A1 = nil, A2 = S1"", A3 = nil
return from process-if(comp(S1""))
region R3 is not modified
Eventually we get a list of 3 regions: R1', R2', R3

5 The PIAF environment

PIAF [5] ([8],[9]) is a FORTRAN programming environment built on the CENTAUR system [6], [5]; it is an OSF/MOTIF application; besides the parallelizer (derived from a preexisting vectorizer: VATIL [14]) it offers facitilities for:

- pretty printing of FORTRAN programs, including the possibility to use different fonts or colors for keywords;
- edition of FORTRAN programs, either by direct edition (cut and paste), or by a link to the Emacs editor;
- display informations gathered on variables, for instance aliases created via EQUIVALENCE statements;
- verifications for conformity with the ANSI standard FORTRAN 77.

The automatic parallelizer performs classical source to source transformations, at the moment on simply nested loops, either on the whole program or on one selected loop; then several actions may be proposed to the user:

- browse the transformed program, highlighting the correspondance with the original program;
- show the dependence graph of some loop;
- allow the user to insert directives which introduce predicates on variables for better dependence computing.

As an illustration we show snapshots of a PIAF session in joint figures: figure 19 shows a window containing the original program of the user, and figure 20 shows the transformed program.

[5] Parallélisation Interactive Automatique de FORTRAN

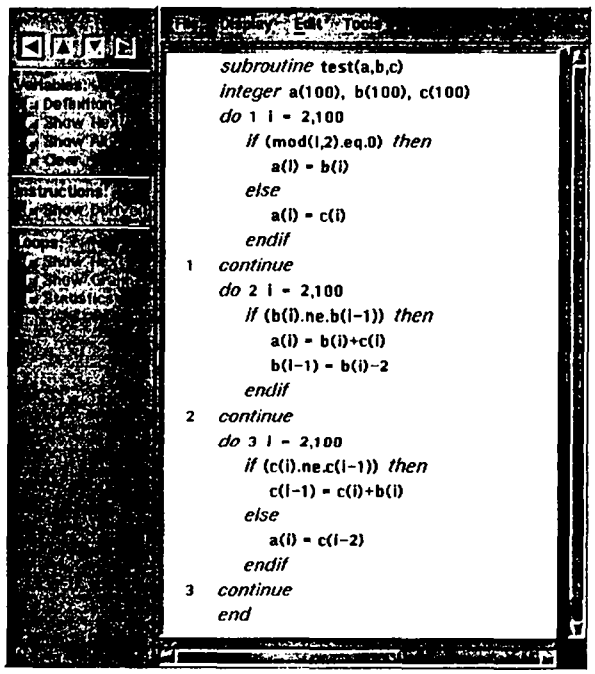


Figure 19: CENTAUR window with original program

```
C$PARALLEL_SECTION
C$DO_PARALLEL 12001
C$LOCAL i
      do 12001 i = 2,100,1
         if (mod(i,2).eq.0) then
            a(i) = b(i)
         else
            a(i) = c(i)
         endif
12001 continue
C$END_PARALLEL_SECTION
      continue
C$PARALLEL_SECTION
C$DO_PARALLEL 12002
C$LOCAL i
      do 12002 i = 2,100,1
         if (b(i).ne.b(i-1)) then
            a(i) = b(i)+c(i)
         endif
12002 continue
C$DO_SEQUENTIAL 12003,AFTER(12002)
C$LOCAL i
      do 12003 i = 2,100,1
         if (b(i).ne.b(i-1)) then
            b(i-1) = b(i)-2
         endif
12003 continue
C$END_PARALLEL_SECTION
      continue
C$PARALLEL_SECTION
C$DO_SEQUENTIAL 12004
C$LOCAL i
      do 12004 i = 2,100,1
         mask(i) = c(i).ne.c(i-1)
         if (mask(i)) then
            c(i-1) = c(i)+b(i)
         endif
12004 continue
C$DO_PARALLEL 12005,AFTER(12004)
C$LOCAL i
      do 12005 i = 2,100,1
         if (.not.mask(i)) then
            a(i) = c(i-2)
         endif
12005 continue
C$END_PARALLEL_SECTION
      continue
```

Figure 20: CENTAUR window with transformed program

6 Related Work

Kennedy and McKinley [10] describe an algorithm whose purpose is similar to that of present paper; their procedure systematically generates masks with 3 possible values noted: TRUE, FALSE, UNDEFINED[6]; this allows construction of smaller boolean guards as with the present method; however it seems on a first view that the present algorithm generates programs which are closer to the original program of the user; furthermore we feel that it is important to try avoiding to generate masks when it is possible, since masks are likely to induce memory traffic.

7 Conclusions

We have described a method for treatment of IF statements in automatic parallelization; as opposed to the methods of [4], [7], the present method does not require the construction of a control flow graph. Our plans for future work are:

- extension of the algorithm to multiply nested loops; the impact would be much greater than for simple loops, since IF conversion in multiply nested loops has to propagate the boolean condition downwards to embedded loops, as in the following example:

```
DO I = 1, 100
   IF (X(I).GT.0) THEN
      DO J = 1, 100
         A(I,J) = B(I,J) + C(I,J)
      END DO
   ENDIF
END DO
```

  If conversion would square square the number of tests:

```
DO I = 1, 100
  DO J = 1, 100
   IF (X(I).GT.0) A(I,J) = B(I,J) + C(I,J)
  END DO
END DO
```

- generation for other targets, for instance generation of vector instructions (FORTRAN-90); the main difference would be the change in the definition of the incompatibility relation #; besides a WHERE statement (vectorized IF statement) cannot have more than one alternative (only one ELSEWHERE clause), thereby forcing splitting of IFs in order to have at most two alternatives in a WHERE statement.

Achnowledgments: We are indebted to Martin Jourdan, Michel Loyer and Gregory Popovitch for careful reading of this paper and making useful comments.

[6] which may be implemented as integers

References

[1] R.ALLEN, K.KENNEDY, C.PORTERFIELD, "Conversion of control dependence to data dependence ",*10th ACM Symposium on Principles of Prog. Lang.*,Austin, 1983 .

[2] A.V.AHO, J.E.HOPCROFT, J.B.ULLMAN, "Design and Analysis of Computer Algorithms ", *Addison-Wesley*, 1974 .

[3] R.ALLEN, D.CALLAHAN, K.KENNEDY, "Automatic Decomposition of Scientific Programs for Parallel Execution ",*14th ACM Symposium on Principles of Prog. Lang.*,Munich, January 1987 .

[4] W.T.BAXTER, "Vectorizing Sequential Loops and Related Optimizations ",*Master's Thesis, University of Wyoming*,August 1988 .

[5] P. BORRAS, D. CLEMENT, T. DESPEYROUX, J. INCERPI, G. KAHN, B. LANG, V. PASCUAL, "CENTAUR: the System ",*proc. of SIGSOFT'88, 3rd Annual Symposium Development Environments, Boston, USA*,1988 .

[6] D. CLEMENT, "GIPE: Generation of Interactive Programming Environments ",*TSI, Vol. 9, N0. 2, 1990, 157 - 165* ,1990 .

[7] J.FERRANTE, K.J.OTTENSTEIN, J.D.WARREN, "The program dependence graph and its use in optimization ",*IBM Research Report, RC 10208*,1983 .

[8] M.C. GIBOULOT, M. LOYER, G. POPOVITCH, F. THOMASSET, "An interactive parallelizer under the CENTAUR environment ",*ESPRIT-II document, project GIPE-2*,1990 .

[9] M.C. GIBOULOT, G. POPOVITCH, F. THOMASSET, "PIAF: User's Guide ",*ESPRIT-II document, project GIPE-2*,1990 .

[10] K. KENNEDY, K.S. MC KINLEY, "Loop Distribution with Arbitrary Control Flow ",*Supercomputing, IEEE, 407-416*,1990 .

[11] D.J. KUCK, R.H. KUHN, D.A. PADUA, B. LEASURE, M. WOLFE, "Dependence graphs and compiler optimizations ",*Proc. 8th. ACM Symp. on Principles of Programming Languages, Williamsburg*,1981 .

[12] C. LAURENT-VERLEYE, "Vectorisation automatique de boucles comportant des instructions de branchement ",*Thèse de Doctorat, Université Paris VI*,28 Septembre 1990 .

[13] M.C. GIBOULOT, E.R. LEBON, M. LOYER, G. POPOVITCH, H. SHAFIE, F. THOMASSET, "Parallel execution of Fortran Programs on the EWS Workstation ",*in Computing Methods in Applied Sciences and Engineering*,R.Glowinski and A.Lichnewsky eds, SIAM, 1990 .

[14] A. LICHNEWSKY, F. THOMASSET, "Techniques de base sur l'exploitation automatique du parallélisme dans les programmes ",*RR INRIA-Rocquencourt n0 460*,Dec 1985 .

[15] M. WOLFE, "Optimizing supercompilers for supercomputers ",*MIT Press*,1989 .

[16] H. ZIMA, B. CHAPMAN, "Supercompilers for parallel and vector computers ",*ACM press*,1991 .

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399