DIRECT PARALLELIZATION OF CALL STATEMENTS

Rémi TRIOLET

University of Illinois
Center for Supercomputing R & D
U.S.A

François IRIGOIN

Ecole des Mines de Paris
Laboratoire CAI
FRANCE

Paul FEAUTRIER

Université PARIS VI
Laboratoire MASI
FRANCE

ABSTRACT

Asynchronous CALL statements are necessary in order to use more than one processor in current multiprocessors. Detecting CALL statements that may be executed in parallel is one way to fill this need. This approach requires accurate approximations of called procedure effects. This is achieved by using new objects called *Region* and *Execution Context*. An algorithm to find asynchronous CALL statements is given. It involves a new dependence test to compute data dependence graphs, which provides better results than previous ones even when no CALL statements are involved. This method has been implemented in Parafrase and preliminary results are encouraging.

Introduction

Supercomputers are now widely used for scientific computations. However, programs need to be restructured to be efficiently executed. As this is a tedious task, automatic tools have been developed that perform a fine grain parallelization.

A new problem arises with currently available multiprocessors (HEP, Cray-XMP, Cray-2, IBM-3090, etc...) since the only way to access their multitasking facilities is to use *asynchronous* CALL statements (CALLs), which create new tasks to execute the called procedures. As a consequence, programs must contain such CALLs to execute on more than one processor. This paper deals with the problem of automatically finding asynchronous CALL statements in Fortran programs.

One solution is to transform existing CALLs into asynchronous CALLs when possible. This solution has proven successful when applied by hand and we present here a method to automatize this process. We call it *Direct Parallelization* of CALLs so as to avoid any confusion with parallelization of CALLs by expansion.

In the first part, the principle of direct parallelization of CALLs is presented and its practical interest is shown. Problems raised by this approach are enumerated in the second part; the most difficult one is to describe parts of arrays accessed [†] by procedure call executions. Two new concepts to deal with these problems, *execution contexts* and *regions*, are introduced in the third part. Then, an interprocedural parallelization algorithm is given and its main steps are explained. The new dependence test, required by this algorithm, is a critical piece and it is discussed in the fifth part.

Some knowledge of parallelization techniques is assumed throughout this paper, see [20] for more details. In the following, calls to functions and subroutines are referred as CALLs. The program is composed of several procedures whose source code is assumed available.

[†] In the following, *accessed* stands for *read or written*.

1. Direct Parallelization of CALL Statements

1.1. Principle

There are four main steps in the process of transforming CALLs into asynchronous CALLs. The first two steps are interdependent but we will see later in which order they are applied to the different procedures.

Statement Effect Computation (SEC). Effects of ordinary statements are computed, as usual, by scanning references to variables. If the procedure contains a few CALLs, their effects are computed by translating results of step *PEC*, applied to the called procedures; variables of these procedures are transformed into variables of the calling ones.

Procedure Effect Computation (PEC). Global effects of a procedure are computed by gathering results of step *SEC* applied to its statements. All accessed variables or parts of variable are associated with the procedure.

Dependence Graph Computation (DGC). The Data Dependence Graph (DDG) can be computed from step *SEC* results, since sets of accessed variables are now attached to CALLs as well as to ordinary statements.

Restructuring and Parallelism Detection (RPD) Then, the program can be restructured and the parallelism can be detected with usual DDG based methods, CALLS being processed like other statements.

This principle is simple, but we have to compute very accurate effects with steps *PEC* and *SEC*, despite the gatherings and translations involved, and then to compute a good DDG at step *DGC*. Otherwise, little parallelism will be found in step *RPD*.

1.2. Advantages of Direct Parallelization

Direct parallelization has interesting features for users, time-space complexity of restructuring compilers and parallelism detection.

As program structure is preserved, users will be able to easily analyze results of source-to-source restructuring compilers. This is useful when users want to know what has been or has not been parallelized in their programs, and when they have to debug them.

Neither the program size nor any subroutine size are changed and procedures are processed in turn. Thus the total amount of computation increases linearly with the number of procedures and not with the number of CALLs. Moreover, at any given time the restructuring compiler has a usual and limited amount of code to process. This is not true with the common solution to handle CALLs: the procedure expansion [9].

Subroutines are not randomly defined. They usually perform sub-computations on subsets of data. Frequent cases of such subsets in scientific programs are rows and columns of matrices or more general parts like diagonals, triangular upper or lower parts, etc... We expect parts of these sub-computations to be executable in parallel (see next section).

No computation, no matter how complex it may be, involving local dynamic arrays of the called procedure will hinder parallelization in step *RPD*. Although they are taken into account in step *PEC*, translation in step *SEC* eliminates their effects. Again, this is not true if the expansion method is used.

1.3. Experiments

Dongarra rewrote standard linear algebra procedures so as to execute CALLs to lower level modules in parallel [5]. Using a HEP computer with one Process Execution Module, he obtained speedups varying from 6.84 to 8.53. The maximum speed-up of this machine being 10, these results are good.

We studied by hand 35 out of 58 subroutines of the LINPACK subroutine library (subroutines with COMPLEX parameters were not considered in this study). This library is especially interesting since approximately 50% of the loops contain one or several CALLs. Our study showed that at least† 27% of these loops could be automatically restructured and parallelized into DOALL [20] or DOACCROSS [4] loops, provided an accurate DDG were computed. For instance, loop 170 in figure 1 extracted from the LINPACK subroutine SSIFA is such a loop. The techniques presented in this paper enable us to discover this parallelism.

```
      DO 170 JJ = 1,K-2
            J = K-1-JJ
            BK = A(J,K)/A(K-1,K)
            BKM1 = A(J,K-1)/A(K-1,K)
            MULK = (AKM1*BK-BKM1)/DENOM
            MULKM1 = (AK*BKM1-BK)/DENOM
            T = MULK
            CALL SAXPY(J,T,A(1,K),1,A(1,J),1)
            T = MULKM1
            CALL SAXPY(J,T,A(1,K-1),1,A(1,J),1)
            A(J,K) = MULK
            A(J,K-1) = MULKM1
  170   CONTINUE

  where SAXPY(N,SA,SX,1,SY,1) is equivalent to:
            SY(1:N) = SY(1:N) + SA*SX(1:N)
```

Figure 1. A Parallel Loop of LINPACK

† A few loops were too large to be analyzed by hand, so they were assumed serial.

2. Problems

We present here in more detail the problems encountered in the different steps. We assume for the sake of simplicity that there are no EQUIVALENCEs, no aliasing and that each COMMON has the same shape in all procedures. All these technical difficulties are dealt with in [17].

2.1. Statement and Procedure Effect Computation (SEC & PEC)

Variables that are read or written by procedures or statements can be enumerated. But, for a given array, the set of accessed elements may be very large and even unknown at compile-time; so, this set must be approximated. Consider for instance the two DO-loops of figure 2; their effects cannot be described with array element lists since index ranges are too big (loop L1) or unknown (loop L2).

```
      DO 10 I = 1,10000        (L1)
         R(I) = S(I)*T(I) + U(I)
10    CONTINUE

      DO 10 I = LB,UB (L2)
         V(I) = 0.0
10    CONTINUE
```

Figure 2.

Different approximations are discussed and a new object called *region* is proposed in § 3. A good approximation cannot be computed unless some knowledge, later called *execution context*, on possible values of variables is available and used in regions. We now have to associate region lists and execution contexts to procedures and statements.

In step *SEC*, basic regions have to be derived from variable references like I or T(I,J) in order to provide homogeneous data to step *DGC*. For a CALL statement, regions associated by step *PEC* to the called procedure, say Q, are related to variables of Q. They have to be transformed into regions related to variables of the procedure containing the CALL. Step *SEC* is detailed in § 4.2.

In step *PEC*, regions attached to statements are gathered to produce regions attached to the whole procedure. This must be done carefully so as to keep the accuracy provided in step *SEC* by execution contexts. Algorithms to associate sets of regions to procedures are discussed in § 4.3.

2.2. Dependence Graph Computation (DGC)

Dependences between statements are usually computed by testing if array references overlap [13]. We have to extend this test to regions, that is, to provide an intersection algorithm for regions. Solutions to this problem are detailed in [17] and presented in § 4.4.

2.3. Restructuring and Parallelism Detection (RPD)

The restructuring process is not deeply changed because application of transformations is driven by the DDG [20]. Consider for instance the loop in figure 1; the decision to apply *scalar renaming* and *scalar expansion* on T or *scalar forward substitution* on BK or MULK does not depend on the nature of the loop body's statements but on the accesses done on these scalars. Maybe a few low-level transformations will have to be modified to cope with CALL statement syntax. This issue is no longer discussed here since it depends on the restructuring compiler used.

The parallelism detection process is entirely driven by the DDG, and hence no changes have to be made. However, once CALLs have been transformed into asynchronous CALLs, synchronization statements have to be added. This is machine dependent and not discussed here.

3. Introduction of Execution Contexts and Regions

A region approximates a set of array elements. This approximation is enhanced if we have some information on the values of variables, especially integer scalars. We call such information an execution context.

3.1. Possible Solutions for a Region

Approximating any array part by the array itself leads to *Summary Data Flow Information (SDFI)* [1]. This solution does not provide much parallelism although Huson proposes it in [9]. Our experiment on LINPACK shows that only 10% instead of 27% of the loops would be parallelized with the SDFI.

In a few cases, it is possible to translate array references from the called procedure into array references of the calling procedure. Conditions on subscript expressions are so stringent that we do not consider this solution any further.

An array part may be described by a range of possible values for each index. The easiest solution is to use integer constants as bounds since such parts are easy to intersect and to translate. This can be extended by accepting symbolic bounds, in which case intersection and translation require symbolic computation.

Loop L in subroutine MM from figure 3 is kept serial with any solution but the last one. Symbolic intervals seem to be interesting but scientific programs use complex parts of arrays that cannot be closely approximated: diagonal, upper part of a matrix, and so on. Regions solve this last problem.

Execution context and region are now formally defined in that order since the later uses the former.

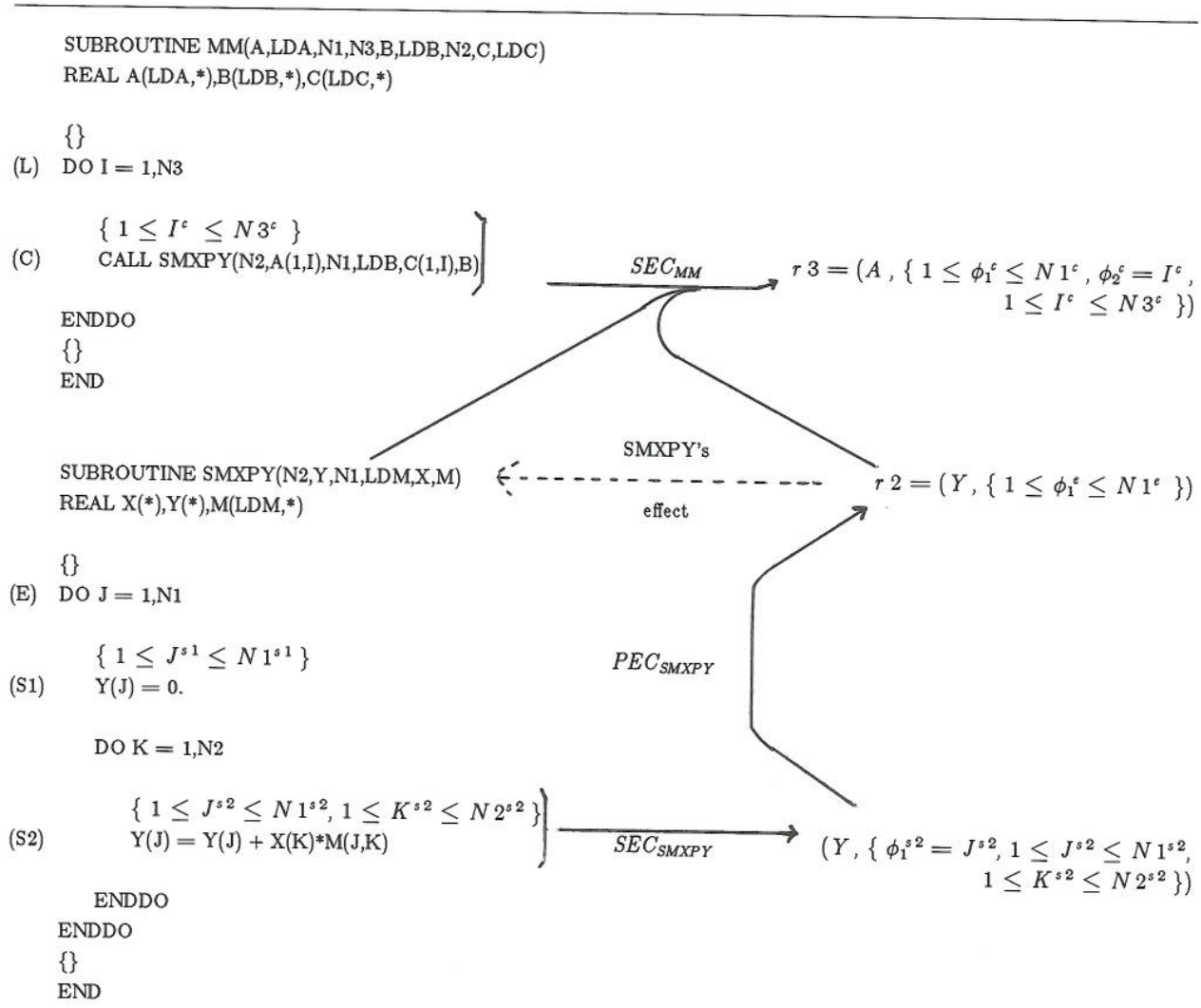

Figure 3. Overview of the Method

3.2. Definition of an Execution Context

Different kinds of information about values of scalar variables in a program can be computed by a *Semantic Analysis*. *Linear inequalities among variables* were chosen because they often provide enough information to parallelize loops and because algorithms from the convex polyhedral theory [7] let us transform symbolic computations into numerical matrix computations.

Definition: Let S be a statement and let s denote one iteration of S. Let V be the set of integer scalars of the procedure and v an element of V. Let v^s be a symbol to denote the value of v before the iteration s and V^s be the set of v^s for all v in V. An execution context attached to s is a system of linear inequalities over V^s.

Example: Here is the execution context for the iteration s of statement S in figure 4.

$$\{ 2 \leq I^s \leq N^s - 1,\ I^s - 1 \leq J^s \leq I^s + 1 \}$$

Other examples of execution contexts are given in figure 3.

```
      DO 10 I = 2,N-1
         DO 10 J = I-1,I+1
            MAT(I,J) = 0.   (S)
10    CONTINUE
```

Figure 4. Two Triangular DO-loops

3.3. Computation of an Execution Context

Several methods have been proposed: *Constant Propagation* [12], *Linear Restraints Among Variables* [3], *Affine Relationship Among Variables* [10]. These intraprocedural methods can be extended to the interprocedural case with only one analysis per procedure [17]. Good results are obtained when one is ready to pay the necessary CPU time.

However simpler methods can provide enough information when dealing with scientific program parallelization. Under a few conditions†, a loop index is constrained by the lower and upper bounds, and this is true over the entire loop body for any of its iterations. Provided these bounds are linear expressions over scalar integer variables, two inequalities may be attached to the loop. Such a loop is called a *regular loop*.

The execution context of a statement is formed by combining all inequalities of the regular enclosing loops. The previous example was computed this way as well as all execution contexts intersparsed among statements in figure 3.

The *regular loop* method and *interprocedural constant propagation* have been implemented in Parafrase [15]. Experiments on LINPACK show that 95% of its loops are regular.

3.4. Definition of a Region

Region attached to an iteration s of a stmt. S

The elements of a d-dimension array T accessed by a reference in s are given by constraining the possible values of T's indices. Let ϕ_i^s, $i \in [1,d]$, be d symbols for these values such that:

$$\Phi^s \bigcap V^s = \emptyset \text{ where } \Phi^s = (\phi_i^s)_{i \in [1,d]}$$

In a region, the possible values of ϕ_i^s are determined by a set of constraints involving the *linear* subscript expressions of the reference and the execution context of s.

Definition 1: a region r of an array T accessed in s is a pair (T,Σ) where Σ is a system of linear inequalities over $V^s \bigcup \Phi^s$. (See region r1 in the next example).

†These conditions are developed in [17]. Their computation requires SDFI.

Region attached to a procedure Q

Let Q be a procedure with one entry point E and q be one execution of Q. Let T be an array referenced many times in Q. The part of T that is accessed by q may be approximated by a region. In order to be meaningful, possible values of indices must be constrained with symbols representing the initial values of all parameters (in a general sense: common and formal variables), that is values associated with e. This is the only way to connect values of parameters in q and values of arguments (in a general sense: common variables and real arguments) in the calling routine P.

Definition 2: a region r of an array T accessed in q is a pair (T,Σ) where Σ is a system of linear inequalities over $V^e \bigcup \Phi^e$.

Examples:†

Statement S in figure 4 writes the region r1:

$$(MAT, \{\phi_1^s = I^s, \phi_2^s = J^s, 2 \leq I^s \leq N^s - 1, \quad I^s - 1 \leq J^s \leq I^s + 1\}) \qquad \text{(r1)}$$

Subroutine SMXPY in figure 3 reads and writes the region r2:

$$(Y, \{1 \leq \phi_1^e \leq N1^e\}) \qquad \text{(r2)}$$

Statement C in figure 3 reads and writes the region r3 obtained by translation of r2:

$$(A, \{1 \leq \phi_1^c \leq N1^c, \phi_2^c = I^c, \quad 1 \leq I^c \leq N3^c\}) \qquad \text{(r3)}$$

One can see with these examples that our method is very accurate: the parts of arrays MAT, Y, and A respectively described by r1, r2 and r3 are equal to the parts actually accessed by the corresponding statements or subroutines.

We show how and when steps *SEC* and *PEC* associate regions to statements and procedures as part of the parallelization process.

4. Finding Asynchronous CALL Statements in a Procedure

Although our method does not involve intricate algorithms, many details have to be taken into account. Only an overview is given in this paper.

4.1. Main Algorithm

Let us call P the procedure we want to process. We assume that the source code of all procedures called directly or indirectly by P is available. Let us call them Q_i for $i = 1,...$

We assume that execution contexts are available for all statements of P and Q_i, using, for example, one

†In these examples, each equality should be replaced by two inequalities.

of the semantics analysis methods listed in § 3.3.

The *calling graph* is acyclic since Fortran prohibits recursivity. So, we can find an order such that any procedure appears after all the procedures it calls. The *Reverse Post Order (RPO)* is one solution [8].

Algorithm:

FOR EACH Q_i following the RPO **DO**

1. Apply *SEC* to Q_i :

   this provides two sets of read and written regions for any iteration of Q_i 's statements;

2. Apply *PEC* to Q_i :

   this provides two sets of read and written regions for Q_i ;

ENDFOR

3. Apply *SEC* then *DGC* and finally *RPD* to P :

   this provides a parallelized P ;

Remark:

There is no reason to parallelize only P since each Q_i can be parallelized as soon as step *SEC* has been applied to it. So a whole program can be processed by applying to the MAIN procedure a slightly modified algorithm where step 1 is replaced by step 3. For libraries, this algorithm must be applied to all top-level procedures.

Part of this algorithm is exhibited in figure 3 where P = MM and Q_1 = SMXPY.

4.2. Guidelines for SEC

Step *SEC* distinguishes ordinary statements and CALLs. Let P be the procedure to analyze.

Case of an ordinary statement S.

The system of a region equivalent to an array reference accessed by an iteration s of S is built by equating variables ϕ_i^s and corresponding subscript expressions, provided they are *linear combinations* of integer scalars.

More formaly, the array reference $T(x_1, x_2, \cdots, x_d)$ is transformed into the region:

$$(T, \bigcup_{x_i \in X} \{ \phi_i^s = x_i \})$$

where

$X = \{ x_i \mid x_i$ is a linear combination of P 's scalars $\}$

For instance, T(I+2*J+1,I*K) becomes:

$$(T, \{ \phi_1^s = I^s + 2J^s + 1 \}).$$

See also region r1 in the example of § 3.4.

Case of a CALL C

Regions accessed by the called procedure, say Q, have been previously computed and associated to Q by step *PEC* because of the reverse post order. They just need to be translated.

Regions of a formal variable V cannot be accurately translated unless V is declared as its corresponding actual argument or as a part of it, when it is an array. Most associations between formal and actual parameters respect this condition: associations such as scalar/scalar, matrix/matrix, matrix/vector, vector/element, etc... See for instance the associations between B and M or between C(1,I) and X in figure 3.

For such associations, regions of Q are renamed, and their inequalities are translated by using equalities between Q 's formal scalar parameters and C 's actual arguments, and between P 's and Q 's common scalar variables with the same offset. This is only possible under linearity condition. Consider for instance the translation of region $r2$ to region $r3$ in figure 3; the double inequality $1 \leq \phi_1^c \leq N1^c$ comes from the translation of $1 \leq \phi_1^e \leq N1^e$ with the equality $N1^c = N1^e$, deduced from variable associations implied by the CALL C; finally the equality $\phi_2^c = I^c$ comes from the array/sub-array association between C(1,I) and X. Note that all cases of region translation are not that simple.

For other associations, a pessimistic attitude must be adopted: whole arrays are assumed touched. Finally, effects on common variables can be translated in the same way due to the assumptions made on COMMONs in § 2.

Including the execution context.

In both cases (CALL or ordinary statement), the execution context associated to the processed statement is added to the previously obtained systems. This can be done since values of scalar variables respect these inequalities when accesses are performed. For instance, the region r1, given in § 3.4 for figure 4, describes a band around MAT's diagonal only because the execution context of S is included.

4.3. Guidelines for PEC

Let Q be the procedure to process, and E is its entry point. For a given array T, we must find the region accessed by all iterations of all Q 's statements as if Q was reduced to a single complex statement associated with E. In order to do that, we have to transform each region $r^s = (T, \Sigma^s)$ into a region $r^e = (T, \Sigma^e)$; elements of V^s in Σ^s must be either eliminated or replaced by expressions on V^e whenever possible.

For a given scalar variable V, an equality relating v^s and elements of V^e can be obtained with a semantics analysis of Q; more simply, detecting non modified variables between e and s provides equalities

such as $v^s = v^e$. For instance, the equalities $N1^e = N1^{s2}$ and $N2^e = N2^{s2}$ hold in figure 3.

As local scalars have no initial values they are eliminated; however, information provided by execution contexts, included in step *SEC*, is kept because of variable elimination algorithm properties. For instance, region r1 of figure 4 is transformed into:

$$(MAT, \{ 2 \leq \phi_1^s \leq N^s\text{-}1, \phi_1^s\text{-}1 \leq \phi_2^s \leq \phi_1^s\text{+}1 \})$$

assuming N is an input parameter and I and J local variables.

Finally, regions of local variables are just ignored. Then we just need to make the union of the resulting regions that describe parts of the same variable. This is done by computing the *convex hull* of their inequality systems (algorithm in [7]).

4.4. Guidelines for DGC

As we assume there is neither EQUIVALENCEs nor aliasing between variables, two regions $r_1 = (T_1, \Sigma_1)$ and $r_2 = (T_2, \Sigma_2)$ cannot overlap if $T_1 \neq T_2$.

Dependence between two statements A and B is due to region overlapping for some of their iterations a and b. If A and B are enclosed in n nested loops, the dependence test is usually performed for a given *execution ordering vector* Λ whose elements specify relationships between loop indices that must hold for a certain kind of dependence to exist [20]. Given $r^a = (T, \Sigma^a)$, $r^b = (T, \Sigma^b)$ and Λ, a linear system Σ^Δ of inequalities containing all information available is built. If Σ^Δ is not feasible, there is no dependence between A and B for array T and vector Λ.

Σ^Δ is a system over $V^a \bigcup V^b \bigcup \Phi^a \bigcup \Phi^b$ containing Σ^a and Σ^b plus three additional systems Σ^O, Σ^Λ and $\Sigma^=$ detailed below.

Overlapping. This condition provides d equations

$$\{ \phi_i^a = \phi_i^b \}_{i \in [1,d]}$$

necessary for intersection. They are transformed into a set of $2 \times d$ inequalities called Σ^O. See part 3 in the next example.

Execution Ordering. If I is the index of a loop that encloses both A and B, Λ provides a relation $I^a \ \lambda_I \ I^b$ where

$$\lambda_I \in \{ <, \leq, >, \geq, =, ? \}$$

($\lambda_I = ?$ is the always true relation). Since all loop indices belong to V, Λ provides a new system of inequalities (possibly empty) over $V^a \bigcup V^b$, called Σ^Λ. See part 4 in the next example.

Execution Context Invariance. In general, no relation is known between v^a and v^b, when v is not a loop index. However, if v is constant for any iteration of A and B, the equality $v^a = v^b$ is verified. Variables N1, N2 and N3 in figure 3 verify such a condition. A few of these variables are easy to detect: variables that are not at all modified for example. They are very often used in subscript expressions after *forward substitution* and *induction variable removal* transformations have been performed. These transformations remove many local variables and replace them with expressions involving loop indices and constant variables, for which information is available.

These equalities are transformed in inequalities so as to form a third system, $\Sigma^=$. See part 5 in the example below.

Example

Let us consider the CALL C to subroutine SMXPY in figure 3. Testing the output dependence from iteration c_1 of statement C to iteration c_2 leads to the following Σ^Δ system:

1. *region* r^{c1}

$$1 \leq \phi_1^{c1} \leq N1^{c1}$$
$$\phi_2^{c1} = I^{c1}$$
$$1 \leq I^{c1} \leq N3^{c1}$$

2. *region* r^{c2}

$$1 \leq \phi_1^{c2} \leq N1^{c2}$$
$$\phi_2^{c2} = I^{c2}$$
$$1 \leq I^{c2} \leq N3^{c2}$$

3. *overlapping of* r^{c1} and r^{c2}

$$\phi_1^{c1} = \phi_1^{c2}$$
$$\phi_2^{c1} = \phi_2^{c2}$$

4. *execution ordering*

$$I^{c1} < I^{c2}$$

5. *execution context invariance*

$$N3^{c1} = N3^{c2}, \ N1^{c1} = N1^{c2}$$

One can easily see that this system is unfeasible thanks to equalities and inequalities between I^{c1}, I^{c2}, ϕ_1^{c1} and ϕ_1^{c2}. So there is no output dependence from iteration c_1 to iteration c_2 of statement C for array A.

Deciding linear inequalities is time consuming. However many constraints are equations like $x = y$ and are used first to reduce the system size. An efficient implementation would even take them into account during the construction of Σ^Δ. A decision algorithm is described in the next section.

5. Evaluation of the DGC Algorithm

There are many reasons to evaluate the *DGC* algorithm independently of the rest of our method. Existing parallelization methods do not handle CALL statements unless they are in-line expanded, so our method can only be compared at the *DGC* level. In

addition, our *DGC* test can be used in existing restructuring compilers without any application to CALLs; as this test includes execution contexts, it sometimes provides better results than other ones. Finally, as the DDG computation is central in restructuring compilers (for instance, Parafrase spends half of its CPU time computing dependences) and as our test requires symbolic computations, we have to show that execution times can be afforded.

5.1. Parallelization Detection Results

Our test was developed to handle CALLs but it also provides new results for triangular loops and vector statements.

CALL Statements

Although the examples shown in this paper are simple, our method has been successfully applied to more complex programs such as block matrix algorithms. It may fail because of a too simple semantics analysis but it can also be used to check parallel CALLs in parallel programs; numerous conditions must be verified and a programmer may always forget a few of them. For instance, detecting the loop L of figure 3 is parallel implies several dependences such as the one detailed in § 4.4 have to be checked.

Vector Statements

The same kind of check arises with vector languages. Statements S1 and S2 from figure 5 would be declared executable in parallel provided that

$$M(I,1{:}I{-}1) \bigcap M(1{:}I{-}1,I)$$

can be automatically proved empty. This is just what we have to do to handle CALLs.

Vector statement *scalarization* [20] could lead to usual parallelization by current restructuring compilers. Unfortunately they are not likely to find such parallelism. For instance, equivalent loop L in figure 5 is not parallelized by Parafrase.

Triangular Loops

A few parallel loops are declared serial by most supercompilers due to DDG inaccuracy. The previously discussed loop L remains serial because statements S1 and S2 are analyzed without their execution context: $1 \leq J \leq I{-}1$. They are found to be output dependent.

According to Veidenbaum [19], only a few parallel loops are not discovered by current restructuring compilers and the CPU time penalty due to the execution context computation and use may seem worthless. However, an accurate DDG would also improve the whole restructuring process. As an example, DOACROSS delays [4] can be shortened by removing only one dependence.

```
TMP = F(I)
M(I , 1:I-1) = +TMP   (S1)
M(1:I-1 , I) = -TMP (S2)
```

Scalarization and
Loop Fusion

```
TMP = F(I)
DO J = 1,I-1  (L)
      M(I,J) = +TMP
      M(J,I) = -TMP
ENDDO
```

Figure 5. Vector Statement Restructuring

5.2. Computational Cost of DGC; Deciding Linear Inequalities

Several methods to decide linear inequalities have been proposed [16], [6], or may be derived from linear programming methods. The Fourier-Motzkin's method has been implemented and experimented with systems produced by the dependence test described in § 4.4. These systems usually have a special form: each inequality involves only a few variables.

Fourier-Motzkin's method decides a system of linear inequalities by successively eliminating its variables. Variable V elimination is done by pairwise combinations of all inequalities involving V so as to preserve constraints induced by V on other variables. A decision is made when an impossible constraint ($0 \leq -b$, $b \in \mathbf{N}^*$) appears, or when the system is empty. Its main advantage is its simplicity: it is quickly implementable. Its main disadvantage is its theoretical complexity, known to be exponential in time and space. In addition, it works on the real line and cannot tell whether integer solutions exist or not; wrong dependences are sometimes found, which is conservative. However, usual cases are properly handled thanks to an extra computation based on a GCD test [2].

Duffin proposes in [6] several improvements of the Fourrier-Motzkin's method so as to obtain a *practical computational algorithm*. In our case where very sparse inequality systems are involved, experiments show that time complexity is in $O(I.V^2)$, where I is the number of inequalities and V the number of variables, and that the memory space used to store intermediate systems is decreasing. Extended results are available in [18]. Nevertheless, our test seems to be an order of magnitude longer than classical ones in which it could be included as a sub-test for hard cases. Moreover, it could probably be optimized.

5.3. Comparison With Other *DGC* Methods

There has been much work in the area of computing dependence graphs [2], [11], [14] & [20], but none of these methods handles compound statements like CALLs or vector statements. Taking into account the loops' execution context is partially done in Wolfe's and Banerjee's methods since a few cases of false dependences can be detected provided the loops' upper and lower bounds are integer constants. This feature was extended in Kuhn's method to triangular loops, but relations built on variables which are not loop indices are not considered; as a consequence, loop L in figure 5 would be declared serial if I is not a loop index.

However, Banerjee's and Wolfe's methods have other advantages. They are based on the resolution of diophantine equations, and different tests are applied depending on the type of subscript expressions encountered. As a result, these methods are fast and can distinguish between real and integer solutions.

Our test is necessary as far as compound statements are concerned and can be applied to ordinary statements if good results on triangular loops are required. On the contrary, if execution time is a premium, classical tests should be used.

Conclusion

We have presented an interprocedural method to globally parallelize multi-routine programs. Although its practical complexity is not as bad as it seems at first glance, its basic steps, statement effect computation, procedure effect computation and dependence graph computation, involve complex algorithms like symbolic computations and linear programming, whose cost in CPU time and memory space is heavy. As a consequence, it is not advocated as a replacement of previous methods but as a complement, used only when necessary, upon detection of CALLs inside DO-loops or upon detection of triangular loops for instance.

The new dependence test developed for interprocedural dependence graph computation can also be used by classical restructuring compilers when CALLs are ignored, in which case the statement effect computation is simplified.

This method was implemented at the *Center for Supercomputing Research and Development* at the *University of Illinois*. Several passes and low level functions were added to Parafrase: SDFI computation, Execution Context computation, SEC and PEC steps, etc... Finally the dependence test was modified in the DDG computation.

Encouraging preliminary results were obtained for CALLs to BLAS in LINPACK: all parallel loops were automatically detected. However, the limited semantics analysis method that was implemented could not cope with complex expressions (e.g. containing MODULO operators) assigned to variables used in subscripts. So, a few BLAS subroutines had to be hand modified.

For more complex programs, powerful semantics analysis methods which provide more information should be used, but one must be prepared to pay the corresponding computational cost. However, the same kind of information, linear inequalities among scalar variables, should be used since up to now failures due to non linear subscript expressions were rare when whole programs were analyzed. Interprocedural constant propagation and forward substitution usually transform a few variables into numerical constants (e.g. size of blocks in block matrix algorithms). The same effect can be reached by programmers using symbolic constants.

Automatic parallelization is not the only issue. Our method can also be used in a programming environment to check parallel programs containing CALLs and to provide diagnoses about dependences. The new concepts introduced, *regions* and *execution contexts,* are useful for other purposes like global optimization, interprocedural checking, array-bound checking, dead-code elimination among others.

References

[1] F. E. Allen, *Interprocedural Data Flow Analysis*, Proc. of the IFIP Congress, North Holland, (1974)

[2] U. Banerjee, *Speedup of Ordinary Programs*, Report No. UIUCDCS-R-79-989, University of Illinois at Urbana-Champaign, (1979)

[3] P. Cousot, N. Halbwacks, *Automatic Discovery of Linear Restraints among Variables of a Program*, in Proc. of the 5th POPL, (1978)

[4] R. G. Cytron, *Compile-time Scheduling and Optimization for Asynchronous Machines*, Report No. UIUCDCS-R-84-1177, University of Illinois at Urbana-Champaign, (1984)

[5] J. J. Dongarra and R. E. Hiromoto, *A Collection of Parallel Linear Equations Routines for the Denelcor HEP*, Parallel Computing, vol. 1(2), North Holland, (1984)

[6] R. J. Duffin, *On Fourier's Analysis of Linear Inequality Systems*, Mathematical Programming Study 1, North Holland, (1974)

[7] N. Halbwachs, *Automatic Discovery of Linear Relationships among Variables of a Program*, in French: *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*, Thèse 3ème cycle, Université de Grenoble (I.N.P.), (1979)

[8] M. S. Hecht, *Flow Analysis of Computer Programs*, North Holland, (1977)

[9] C. A. Huson, *An In-Line Subroutine Expander for Parafrase*, Rep. UIUCDCS-R-82-1118, University of Illinois at Urbana-Champaign, (1982)

[10] M. Karr, *Affine Relationships among Variables of a Program*, Acta Informatica, No 6, (1976)

[11] K. Kennedy, *Automatic Translation of Fortran Programs to Vector Form*, Report No. 476-029-4, Rice University, (1980)

[12] G. Killdal, *A Unified Approch to Global Program Optimization*, Proc. of the 1st POPL, (1973)

[13] D. J. Kuck, *Dependence Graph and Compiler Optimizations*, Proc. of the 8th POPL, (1981)

[14] R. H. Kuhn, *Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage Networks, and Decision Trees*, Report No. UIUCDCS-R-80-1009, University of Illinois at Urbana-Champaign, (1980)

[15] Analyzer Documentation (PARAFRASE), Center for Supercomputing R & D, University of Illinois at Urbana-Champaign, (1985)

[16] R. Shostak, *Deciding Linear Inequalities by Computing Loop Residues*, ACM Journal, Vol. 28(4), (1981)

[17] R. Triolet, *Contribution to Automatic Parallelization of Fortran Programs with Procedure Calls* in French: *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédure*, Thèse de Docteur-Ingénieur, Université PARIS VI (I.P.), (1984)

[18] R. Triolet, *Interprocedural Analysis Based Restructuring of Fortran Programs*, Proc. of the International Workshop, Parallel Algorithms & Architectures, Marseille (France), to be published by North-Holland, (Apr. 14-18, 1986)

[19] A. Veidenbaum, *Compiler Optimizations and Architecture Design Issues for Multiprocessors*, Report No. UIUCDCS-R-85-1207, University of Illinois at Urbana-Champaign, (1985)

[20] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers*, Report No. UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign, (1982)