HAL
open science

Compiling sequential programs for distributed memory parallel computers with PANDORE II

Françoise André, Olivier Chéron, Jean-Louis Pazat

To cite this version:

Françoise André, Olivier Chéron, Jean-Louis Pazat. Compiling sequential programs for distributed memory parallel computers with PANDORE II. [Research Report] RR-1667, INRIA. 1992. inria-00074890

HAL Id: inria-00074890

https://inria.hal.science/inria-00074890

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-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1992

25ème anniversaire

N° 1667

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

COMPILING SEQUENTIAL PROGRAMS FOR DISTRIBUTED MEMORY PARALLEL COMPUTERS WITH PANDORE II

Françoise ANDRÉ
Olivier CHÉRON
Jean-Louis PAZAT

Avril 1992

RR.1667

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Compiling Sequential Programs for Distributed Memory Parallel Computers with Pandore II

Françoise André, Olivier Chéron, Jean-Louis Pazat *

Publication Interne n° 651 - Avril 1992 - 18 pages

Programme 1

Abstract

Parallelization of programs for Distributed Memory Parallel Computers is always difficult because of many low-level problems due to a programming model based on parallel processes. The PANDORE system has been designed to allow programmers to maintain a sequential programming style.

The PANDORE compiler generates parallel processes according to a data decomposition specified by the programmer. For this purpose, new constructs have been added to an existing sequential language. Code generation relies on the SPMD model and on the *locality of writes* rule. A prototype has shown the feasibility of this approach. The second version of PANDORE we are currently implementing is described here.

We present the language features related to data distribution together with the compiling techniques. The programming methodology is illustrated by the parallelization of the Gram-Schmidt algorithm.

*e-mail: pandore@irisa.fr

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE (U.R.A. 227) UNIVERSITE DE RENNES I I.N.S.A. DE RENNES
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE (UNITE DE RECHERCHE DE RENNES)

Compilation de programmes séquentiels pour machines parallèles à mémoire distribuée avec Pandore II

Françoise André, Olivier Chéron, Jean-Louis Pazat

Programme 1

Résumé

Nous avons conçu le système PANDORE pour répondre au souhait des utilisateurs d'être déchargés des problèmes de bas niveau liés à la parallélisation et à la distribution d'applications sur machines parallèles à mémoire distribuée (MPMD).

PANDORE permet l'exécution parallèle sur MPMD de programmes écrits dans un langage séquentiel. Le compilateur du système PANDORE transforme le programme en processus parallèles à partir d'une décomposition logique des structures de données fournie par l'utilisateur grâce à de nouvelles constructions introduites dans le langage. Le code produit par le compilateur PANDORE est de type SPMD avec écritures locales.

Un premier prototype a permis de montrer la faisabilité de cette approche. Nous présentons ici la deuxième version de ce système : nous décrivons le langage incluant les spécifications de décomposition ainsi que les techniques de compilation employées. Enfin, un exemple (l'algorithme de Gram-Schmidt) illustre la méthode de programmation proposée.

1 Introduction

Presently the most widely used method to program distributed memory parallel computers (DMPC for short), is to insert system library calls inside sequential codes in order to create parallel processes and to express their cooperation using message-based communication.

With this way of programming, the user has to take into account the distribution of the code among processors and the distribution of data onto the distributed memory. Code distribution and data distribution are closely related, leading to very cumbersome programs. For example, implementing data accesses requires the user to write conversions between global and local names of variables, to compute array subscripts and to cope with interprocess communication when it is needed.

Opposite to this programming technique is the wish of most users to be freed from the distributed aspects of their programs which they want also to be machine independent. The goal of the PANDORE system is to allow a parallel execution of programs on DMPC without requiring from the user any deep knowledge of the machine architecture. In PANDORE the user almost keeps following the well-understood sequential model, leaving to the compiler the parallelization and the distribution tasks. Similar approaches are investigated in other projects such as [1, 2, 3].

A first prototype of the PANDORE compiler [4, 5] has been realized and has shown the feasibility of the approach. From this experience, we are currently implementing a second version PANDORE II. The basic concepts remain the same, but some ad-hoc features have been removed; moreover, the compiler has been written in a modular fashion, allowing easy integration of new parallelization and distribution rules as well as implementation of different optimization techniques. The purpose of this paper is to present this new version.

The PANDORE system provides the user with a mean to logically specify data partitioning and mapping in a sequential program in order to execute it on a DMPC. The language aspect is described in section 2.

From such an annotated program, the PANDORE compiler automatically generates parallel distributed processes and the necessary data communication. The compiling technique relies on the SPMD model and the *owner-write* rule; it is described in section 3.

In section 4 we illustrate the programming method with PANDORE II, through the example of the Gram-Schmidt algorithm.

2 The PANDORE Language

The PANDORE language is based on a sequential imperative language. A prototype has been defined using a subset of the C language as a basis. The description of the subset of C is given in 2.2. With similar restrictions it is possible to specify a subset of FORTRAN, so that our technique could easily be applied to FORTRAN.

2.1 Distribution Related Features

A PANDORE program is a sequential program which calls distributed phases. The sequential part is notably in charge of all I/O operations. A distributed phase is spread over the processors of the target DMPC and is executed in parallel. Its specification is described similarly to the definition of a procedure:

- a distributed phase is given a name and a list of formal parameters,
- the occurrence of that name with a list of effective parameters, in the subsequent program text, produces the instantiation of the distributed phase,
- the body of a distributed phase is written as a sequential procedure; there are no parallel constructs in the language,
- a distributed phase cannot call another distributed phase; they are called by the main program.

The statement

dist *d-phase* (*distributed parameter list*) *d-block*

introduces the distributed block of instructions *d-block*.

The distributed parameter list is the main feature of the PANDORE language. It allows to specify the partitioning and the mapping of the data used in the distributed phase.

The array is the only data type which may be partitioned. The mean to decompose an array is to split it into blocks. The specification of the partitioning for a d-dimensional array is given by the keyword **block** $(t_1, ..., t_d)$ where t_i indicates the size of the blocks in the ith dimension. For example

$$Y[NC][MC] \textbf{ by block } (1, MC)$$

indicates that the array Y of $NC \times MC$ elements is decomposed into blocks of size $1 \times MC$: the array is decomposed into NC lines.

Then, the mapping of the blocks onto the architecture will be achieved by the compiler according to the number of processors of the real architecture. This number is given by the user at compile time, using an option in the compiling command line.

In his program, the user has two possibilities to guide the compiler for the mapping: he can either specify that the blocks of a partitioned array are consecutively grouped to be placed on one processor, or specify that the blocks are cyclically wrapped onto the different processors.

The first possibility is expressed through the mapping function

map regular $(x_1, ..., x_d)$

where the vector (x_i) represents a permutation of $(0, 1, ..., d-1)$, where d is the dimension of the array; this permutation indicates to the compiler the way to go through the different dimensions to perform its allocation.

The second possibility is expressed by the function **wrapped** $(x_1, ..., x_d)$ where (x_i) have the same meaning as previously. An example of decomposition and mapping for a 2-dimensional array is shown in figure 1.

The last specification given in the parameter list concerns the transfer `<mode>` for values between the sequential data space $\mathcal{S}$ (data used in the sequential part of the program) and the distributed data space $\mathcal{D}$; allowed modes are **IN, OUT** and **INOUT**:

IN the values of $\mathcal{D}$ are read from $\mathcal{S}$ at the beginning of the distributed phase, but at the end of the phase, the values of $\mathcal{D}$ are not copied back to $\mathcal{S}$,

OUT the values of $\mathcal{D}$ are not initialized from $\mathcal{S}$ at the beginning of the distributed phase, but the values are copied from $\mathcal{D}$ to $\mathcal{S}$ at the end of the phase,

INOUT values are copied from $\mathcal{S}$ to $\mathcal{D}$ at the beginning of the distributed phase, and from $\mathcal{D}$ to $\mathcal{S}$ at the end of the phase.

The value of the `<mode>` parameter allows to know whether it is necessary to distribute (to collect) the actual content of the data structure at the beginning (at the end) of a distributed phase or not.

With this very restricted set of concepts (blocks partitioning and mapping) we are able to obtain most usual decompositions. We think that more complex decompositions are very difficult to express in a language and for a compiler to analyze. So, complex decompositions may lead to inefficient parallel code. Moreover, they are seldom used because it becomes very difficult for the user to understand what will happen during program execution concerning locality and communication. Other proposals for sequential language extensions are submitted, notably in [6, 7]

The PANDORE language has been designed in order to provide the user with a small set of simple and well-defined data distribution features; if other regular decomposition functions reveal to be frequently used, their addition to the language could be envisaged. Nevertheless, our purpose is not to enable a general decomposition scheme like in the BOOSTER language [8]. In case rare decomposition is valuable and needed, the user will probably be able to explicitly code its program using a message-based language.

2.2 The C Subset

With our compiling technique, the data decomposition enforces parallelization. As it will be shown in section 3.3, when regular decompositions are used together with regular data accesses patterns, it is possible to generate very efficient code.

The design of powerful compile-time techniques is the main purpose of our study. The input language has been restricted to a subset of C, without pointers and side-effect functions. Indeed the parallelization rules in the compiler would be of no use for such dynamic aspects of the C language.

The treatment of irregular decompositions could only be done at run time, for example by using the *inspector/executor* technique described in [9]; but this is not our present research interest.

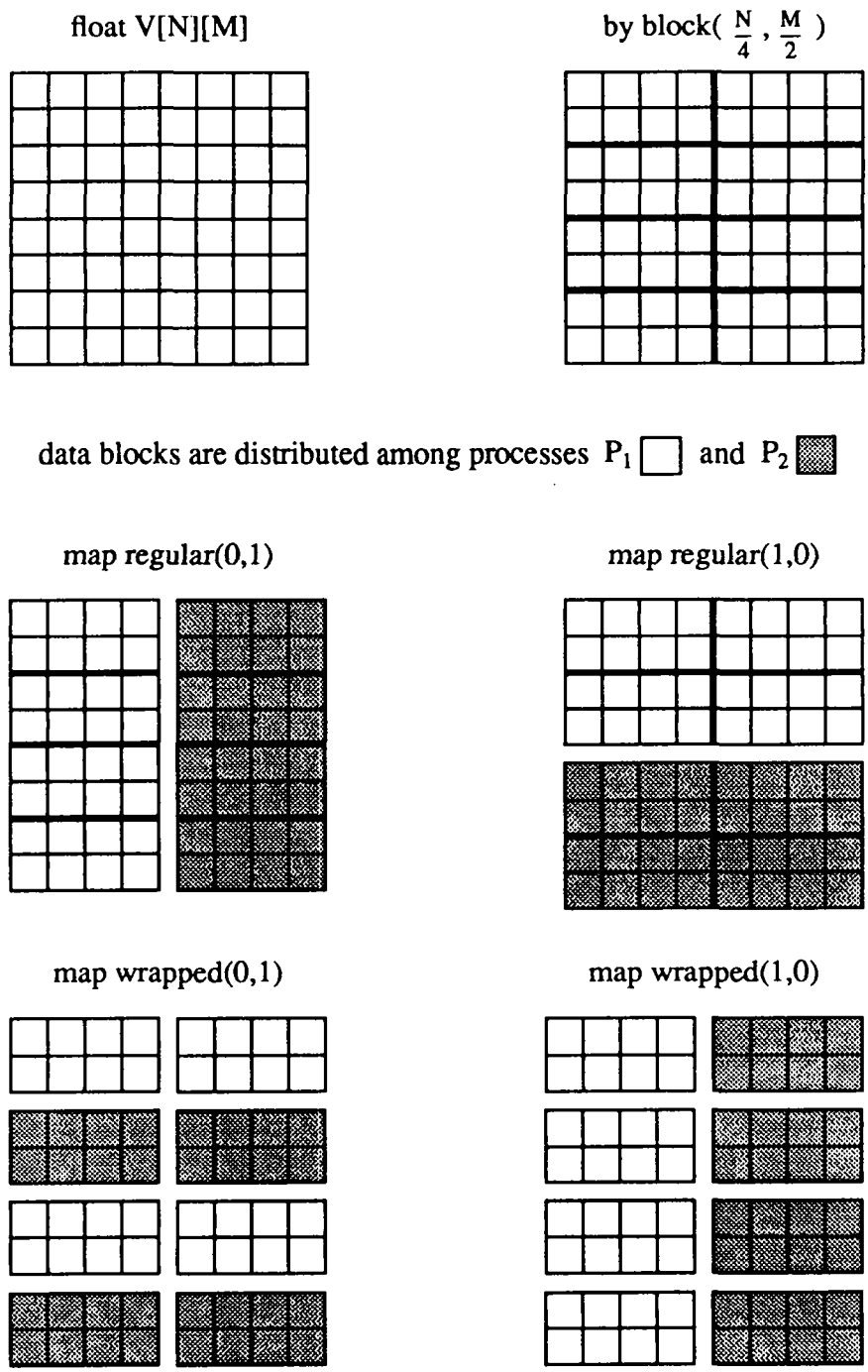

Figure 1: Example of array decomposition and mapping

There are other minor restrictions in the current version of the input language, such as the impossibility to declare structured variables. These restrictions may disappear in future versions of our system. We know how to manage these concepts; but at present it will complexify the compiler without giving new possibilities for the parallelization/distribution aspects.

2.3 Other Language Extensions

Some constructs have been added to the C language, with no direct relation with the distribution aspects, to improve the ease of programming. The two features which are offered to the programmer are *macros* and *closed functions*. *Macros* declarations are similar to procedure definitions. In the program text, a call to a *macro* is in-lined by the compiler. The parameter substitution mode is by name. Direct access to global variables (side-effects) is not allowed. *Closed functions* are similar to C functions but they cannot access global variables. The parameter substitution mode is by value as in C. Meanwhile the *closed functions* may reference global user-defined types of variables, other *closed functions* or library functions.

3 The PANDORE Compilation Scheme

Our system generates a SPMD program by almost replicating the initial code. The compilation scheme is based on the locality of writings. A statement that modifies a variable `v` is executed by the process that stores `v` in its local memory. So when a statement references non-local variables, messages have to be generated in order to obtain the distant values.

3.1 Basic Scheme

To deal with remote accesses, the following protocol has been defined. Let `S` be an assignment (extracted from a distributed block); `S` is compiled into the three intermediate language macros :

- `REFRESH(TS,USE(S),OWN(DEF(S)))`
- `EXEC(OWN(DEF(S)),`$\tilde{S}$`)`
- `FREE(TS)`

where

- `TS` is a set of temporary storage which is allocated by the `REFRESH` macro for the receipt of distant values,
- `USE(S)` is the set of variables read by `S`,
- `DEF(S)` is the set of variables modified by `S`,
- `OWN(X)` is the set of processes data in `X` are mapped onto (`X` being a set of variables),
- $\tilde{S}$ is the `S` statement in which the references to distant variables have been substituted by the names of the corresponding temporary storage.

The aim of the REFRESH macro is to make the values of all the distant variables available locally for the process on which the left hand side variable is located. The EXEC macro is used to mask the assignment in order to execute it only on the processes which are responsible for the management of the assigned variables. The FREE macro frees the temporary storage TS allocated by REFRESH. The compiler uses these three macros to translate each assignment contained in a `dist` block. For example, the assignment `A[i] = B[i] + C[j]` is translated into :

```
REFRESH({tmp1,tmp2}, {B[i],C[j]}, OWN({A[i]})) ;
EXEC(OWN({A[i]}), A[i]= tmp1 + tmp2);
FREE({tmp1,tmp2})
```

In this example, the REFRESH macro will cause the owners of `B[i]` and `C[j]` to send their values to the owner of `A[i]`, and the owner of `A[i]` to receive the value of `B[i]` in tmp_1 and the value of `C[j]` in tmp_2. The EXEC macro ensures that the owner of `A[i]` will be the only process to perform the assignment `A[i]`$= tmp_1 + tmp_2$.

A straightforward run-time implementation of the REFRESH, EXEC and FREE macros is shown below:

```
REFRESH(TS,U,P)        ≡ foreach v ∈ U do
                             TS = TS⋃allocate(tmp);
                             refreshvar(tmp,v,P)
                         endforeach

refreshvar(tmp,v,P)    ≡ if myself ∈ OWN({v})
                             then send(P\OWN({v}), {v})
                         if myself ∈ P\OWN({v})
                             then recv(OWN({v}),tmp)

EXEC(S̃,P)              ≡ if myself ∈ P
                             then S̃

FREE(TS)               ≡ foreach tmp ∈ TS do
                             free(tmp)
                         endforeach
```

where

- `myself` is the name of the current process,
- `allocate(tmp)` allocates a temporary storage that will be referenced by "`tmp`",
- `free(tmp)` frees the temporary storage `tmp`,
- `send(Q,W)` sends the values of the variables in `W` to the processes in `Q` ("non-blocking send": processes in `OWN(W)` do not wait for values in `W` to be received on `Q`),
- `recv(Q,W)` waits for receiving values in the temporary storage set `W` from processes in `Q`,
- communication between processes is FIFO, no messages are lost.

A clever method to implement the `allocate/FREE` macros is to use the stack of the target machine instead of the heap. This could be achieved by merely opening/closing a block if the target language allows block structures.

Let us see how our example is translated, assuming that all the values are integers and that `A[i]` is located on the process P_1, `B[i]` is on P_2 and `C[j]` is on P_3 :

```
{int tmp1, tmp2 ;

 if myself ∈ {P2}
     then send({P1},{B[i]}) ;
 if myself ∈ {P1}
     then recv({P2},{tmp1}) ;

 if myself ∈ {P3}
     then send({P1},{C[j]}) ;
 if myself ∈ {P1}
     then recv({P3},{tmp2}) ;

 if myself ∈ {P1}
     then A[i]= tmp1 + tmp2 ;
}
```

3.2 Conditional Statement

The compilation of a conditional statement (**if** `C` **then** `S`) is less straightforward. The processes involved in the evaluation of such a statement are not solely the owners of the variables referenced in `S`, but also the owners of the variables necessary for the computation of the condition `C`. A conditional statement found in a distributed phase is compiled as shown below:

```
if myself ∈ OWN(USE(C)∪USE(S)∪DEF(S))
  then  foreach v ∈ USE(C) do
            allocate(tmp);
            refreshvar(tmp,v,OWN(USE(C)∪USE(S)∪DEF(S)))
        endforeach
        if C̃ then s
```

Each variable referenced by `C`[1] has to be refreshed on the processes responsible for the evaluation of the **if** statement (processes that belong to `OWN(USE(C)∪USE(S)∪DEF(S))`). Then, the condition $\tilde{C}$ is evaluated in order to fix whether the statement `s` is to be executed (`s` represents the compilation of `S`) or not.

3.3 A Framework for Loop Optimization

Most of the inherent parallelism of a given program can be found in loops. An efficient parallel code must take into account the data distribution at the loop level, in order to

[1] `USE(C)` is the set of variables referenced by the expression `C`

remove unnecessary operations such as the REFRESH/EXEC statements which generate no data exchange. This can be achieved at compile time by the analysis of iteration domains. The compiler has to divide these domains into different subdomains:

- those involving only local computations, and
- the subdomains in which computations need interprocess communication.

Domain analysis necessitates the use of linear programming techniques [10] such as the Chernikova algorithm [11]. Our system provides hooks to easily interface the PANDORE compiler with such algorithms. By using domain analysis, the efficiency of the produced code can be mainly improved in the two following ways:

- restriction of the code of each process, and
- communication reorganization.

Indeed, a process should execute only the statements referencing data it is truly responsible for, and synchronization with other processes could be limited by anticipating some variable refreshments.

Consider for example the following program fragment:

```
int A[N]    by block(N/P)   map wrapped(0)
int B[N][N] by block(N/P,N) map regular(0,1)
int C[N][N] by block(N/P,N) map regular(0,1)
...
for(i=1;i<N;i++) {
   A[i]=f(A[i]);
   for(j=1;j<N;j++)
      B[i][j]=A[i]*C[i][j];
}
```

In this example, A[i] is refreshed by each instance of B[i][j]=A[i]*C[i][j]. As A[i] is not modified by the body of the j-loop, its refreshment can be moved before the beginning of each j-loop. Moreover, if the compiler can detect that arrays B and C are partitioned the same way, it can omit refreshment of every C[i]. So, once A[i] has been refreshed, the assignment of the inner loop can be performed locally with no further variable refreshment. The iteration space of the inner loop can also be restricted on each process so that j covers only the B[i][j] mapped onto the process. This code improvement can be achieved combining dependence and domain analysis.

However, depending on the shape of the different data and computation domains, computing the intersection of these domains at compile time is not always possible. When the domains are very irregular, most variable refreshments cannot be removed.

3.4 The Compiler Developing Environment

Our compiler has been written using the CAML [12] environment. CAML is a functional language (based on the ML language) providing polymorphic type synthesis, pattern matching and facilities to define parsing functions. Using these features, a *standard* intermediate form on which the compiler works has been defined. This *standard* form is independent of the

source language, so that our compiling techniques could be applied to any imperative sequential languages with partitioning capabilities. Moreover, the power of CAML allows us to easily define complex data structures that are needed to handle compile-time optimizations, and helps the building of the compiler itself in a modular way.

4 Example: The Modified Gram-Schmidt Algorithm

4.1 The Algorithm

Given a set of independent vectors $\{v_1, ..., v_n\}$ in R^m, the Modified Gram-Schmidt algorithm produces an orthonormal basis of the space generated by these vectors [13]. The basis is constructed step by step, each new computed vector replaces the old one. This algorithm can be simplified by using the following notation (figure 2). $[i, i]$ means that vector i is normalized ($v_i = v_i/\|v_i\|_2$) and $[i, j]$ means that vector j is corrected with vector i ($v_j = v_j - (v_i^T.v_j) * v_i$). This algorithm has potential medium-grain parallelism. Indeed, each correction (i.e. $[i, j]$) can be computed in parallel. Since each of them has the same number of computations, load balancing can be easily achieved.

```
for i = 1 : n
    (* Normalization *)
    [i, i]
    for j = i + 1 : n
        (* Correction *)
        [i, j]
    end
end
```

Figure 2: Simplified MGS algorithm.

4.2 The PANDORE Approach

The user should first write the sequential program shown in figure 3. In order to execute this code in parallel with PANDORE II, the user has to distribute the data (v_i) represented by Y[i][0..MC] among processors as the compiler uses the *local-write* scheme to compile the code.

The simplest idea is to divide this array in blocks of size NC/P $\times$ MC, and to map each block on a different processor. This leads to the code shown in figure 4.

```
#define NC 1024
#define MC 1024

float Y[NC][MC] ;
float xnorm[NC] ;
float sdot[NC] ;
int i,j,k ;

main()
{
  /* initializing vectors */
  ....
  /* computing vectors */
  for (i=0; i<NC; i++) {
    /* normalization */
    xnorm[i] = 0.0 ;
    for (k=0; k<MC; k++)
      xnorm[i] = xnorm[i] + Y[i][k]*Y[i][k] ;
    xnorm[i] = 1.0/sqrt(xnorm[i]) ;
    for (k=0; k<MC; k++)
      Y[i][k]=Y[i][k]*xnorm[i] ;
    /* correction */
    for (j=i+1; j<NC; j++) {
      sdot[j]=0.0 ;
      for (k=0; k<MC; k++)
        sdot[j]=sdot[j]+Y[i][k]*Y[j][k] ;
      for (k=0; k<MC; k++)
        Y[j][k]=Y[j][k] - sdot[j]*Y[i][k] ;
    }
  }
}
```

Figure 3: Sequential MGS algorithm.

```
#define P 64
#define NC 1024
#define MC 1024

float V[NC][MC] ;

dist vect(float Y[NC][MC] by block(NC/P,MC) map regular(0,1) mode INOUT)
float xnorm[NC] by block(NC/P) map regular(0);
float sdot[NC] by block(NC/P) map regular(0);
{
  int i,j,k ;
  /* computing vectors */
  for (i=0 ; i<NC; i++) {
    /* normalization */
    xnorm[i] = 0.0 ;
    for (k=0; k<MC; k++)
      xnorm[i] = xnorm[i] + Y[i][k]*Y[i][k] ;
    xnorm[i] = 1.0/sqrt(xnorm[i]) ;
    for (k=0; k<MC; k++)
      Y[i][k]=Y[i][k]*xnorm[i] ;
    /* correction */
    for (j=i+1; j<NC; j++) {
      sdot[j]=0.0 ;
      for (k=0; k<MC; k++)
        sdot[j]=sdot[j]+Y[i][k]*Y[j][k] ;
      for (k=0 ;k<MC; k++)
        Y[j][k]=Y[j][k] - sdot[j]*Y[i][k] ;
    }
  }
main ()
{
/* initializing vectors */
...

vect(V);
}
```

Figure 4: Parallel MGS algorithm using PANDORE

This code is very similar to the sequential one and there is no need to modify the description of the computation to take into account the data distribution. The fundamental instructions remain unchanged: the programmer familiar with the C language will have no problem to program his algorithm for the PANDORE compiler.

The main difference between the two versions of the algorithm is the use of the **dist** construct, syntactically very similar to a function. Data used both in the distributed computation and in sequential parts of the program are parameters of the **dist** construct. The partitioning and distribution are added to the type of parameters in the declaration of the **dist** construct.

Arrays like **sdot** and **xnorm** are local to the distributed part of the computation, so they are not in the parameter list and have no mode (IN/OUT or INOUT).

However, this code is not very efficient because the correction is different on each vector: vector v_i is corrected i times, so it is not a good idea to distribute the matrix Y by blocks.

A better load balancing will be achieved if the matrix is partitioned in lines mapped in a wrapped fashion onto the processors. To do this, the user has only to change the following lines :

```
dist vect(float Y[NC][MC] by block(NC/P,MC) map regular(0,1) mode INOUT)
float xnorm[NC] by block(NC/P) map regular(0);
float sdot[NC] by block(NC/P) map regular(0);
into
dist vect(float Y[NC][MC] by block(1,MC) map wrapped(0,1) mode INOUT)
float xnorm[NC] by block(1) map wrapped(0);
float sdot[NC] by block(1) map wrapped(0);
```

As vector v_i is used for the corrections of vectors v_j for each $j > i$, the use of an intermediate replicated vector vc will avoid multiple distant accesses to v_i. The final code is shown in figure 5.

5 Conclusion

With the restrictions mentioned earlier, the PANDORE II compiler is now working. For instance, the code shown in figure 5 has been compiled as such. We are still designing and implementing new optimizations. As said in 3.3, the compiler has been built in such a manner that all the information necessary for optimization purposes are clearly identified. Optimization rules may be added step by step, without changing the structure of the compiler.

We hope to have shown in this paper that, thank to our parallelization and distribution technique, the user is free from the burden of processes and communication management. The program which is intended to run on a DMPC is very similar to a sequential program. Moreover the user may easily experiment different partitioning strategies without having to rewrite his whole program. In our case, the computation part remains the same. The situation is very different with a hand-coded program using message-based operations: if the user wants to change the data distribution, he will have to deeply modify his program as all data exchanges will be altered.

```
#define NC 1024
#define MC 1024

float V[NC][MC] ;

dist vect(float Y[NC][MC] by block(1,MC) map wrapped(0,1) mode INOUT)
float xnorm[NC] by block(1) map wrapped(0);
float sdot[NC] by block(1) map wrapped(0);
{
  int i,j,k ;
  float vc[MC]; /* replicated on each processor */
  /* computing vectors */
  for (i=0; i<NC; i++) {
    /* normalization */
    xnorm[i] = 0.0 ;
    for (k=0 ;k<MC; k++)
      xnorm[i] = xnorm[i] + Y[i][k]*Y[i][k] ;
    xnorm[i] = 1.0/sqrt(xnorm[i]) ;
    for (k=0; k<MC; k++)
      Y[i][k]=Y[i][k]*xnorm[i] ;
    /* global update */
    for (k=0; k<MC; k++)
      vc[k]=Y[i][k] ;
    /* correction */
    for (j=i+1; j<NC; j++) {
      sdot[j]=0.0 ;
      for (k=0; k<MC; k++)
        sdot[j]=sdot[j]+vc[k]*Y[j][k] ;
      for (k=0; k<MC; k++)
        Y[j][k]=Y[j][k] - sdot[j]*vc[k] ;
    }
  }
main ()
{
/* initializing vectors */
...
vect(V);
}
```

Figure 5: Parallel MGS algorithm using PANDORE (optimized version)

References

[1] M. Rosing and R. B. Schnabel. *Efficient Language Constructs for Large Parallel Programs - An Overview of Dino2.* Technical Report CU-CS-578-92, University of Colorado at Boulder, 1992.

[2] H. P. Zima, H.-J. Bast, and M. Gerndt. SUPERB: a tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, (6):1–18, 1988.

[3] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, 1988.

[4] H. Thomas. *Une approche de la compilation de programmes séquentiels pour machines à mémoire distribuée.* PhD thesis, IFSIC/Université de Rennes I, June 1991.

[5] F. André, J.L. Pazat, and H. Thomas. Pandore: A System to Manage Data Distribution. In *International Conference on Supercomputing*, ACM, June 11-15 1990.

[6] S. Hirananandani, K. Kennedy, C. Koelbel, U.Kremer, and C-W. Tseng. *An Overview of Fortran D Programming System.* Technical Report TR91121, CRPC, RICE University, March 1991.

[7] B. Chapman, P. Mehrotra, and H. Zima. *Vienna Fortran: A Fortran Language Extension for Distributed Memory Multiprocessors.* Technical Report 91-72, ICASE, September 1991.

[8] E. M. Paalvast and A. J. Van Gemund. A method for parallel program generation with an application to the *Booster* language. In *International Conference on Supercomputing*, pages 457–469, June 1990.

[9] C. Koelbel and P. Mehrotra. *Supporting Shared Data Structures on Distributed Memory Architectures.* Technical Report csd-tr 915, Department of Computer Science, Purdue University, 1990.

[10] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, 1986.

[11] N.V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. *USSR Computational Mathematics and Mathematical Physics*, 5(2):228–233, 1965.

[12] M.V. Aponte, A. Lavaille, M. Mauny, A. Suarez, and P. Weis. *The CAML Reference Manual.* Technical Report, INRIA, 1990.

[13] G. H. Golub and C. F. Van Loan. *Matrix computations.* The Johns Hopkins University Press, second edition edition, 1990.

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA

PI 642 ARCHE : UN LANGAGE PARALLELE A OBJETS FORTEMENT TYPES
Marc BENVENISTE, Valérie ISSARNY
Mars 1992, 132 pages.

PI 643 CARTESIAN AND SATISTICAL APPROACHES OF THE SATISFIABILITY PROBLEM
Israël-César LERMAN
Mars 1992, 58 pages.

PI 648 SET-THEORETIC GRAPH REWRITING
Jean-Claude RAOULT, Frédéric VOISIN
Mars 1992, 18 pages.

PI 649 UNE STRUCTURE D'INFORMATION POUR LES ALGORITHMES D'EXCLUSION MUTUELLE FONDES SUR UNE ARBORESCENCE
Jean-Michel HELARY, Achour MOSTEFAOUI, Michel RAYNAL
Mars 1992, 18 pages.

PI 650 BLOCK-ARNOLDI AND DAVIDSON METHODS FOR UNSYMMETRIC LARGE EIGENVALUE PROBLEMS
Miloud SADKANE
Avril 1992, 24 pages.

PI 651 COMPILING SEQUENTIAL PROGRAMS FOR DISTRIBUTED MEMORY PARALLEL COMPUTERS WITH PANDORE II
Françoise ANDRE, Olivier CHERON, Jean-Louis PAZAT
Avril 1992, 18 pages.

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

ISSN 0249-6399