HPF to C-Pandore translator

Lahcen Jerid, Françoise André, Olivier Chéron, Jean-Louis Pazat, T. Ernst

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

HPF to C-PANDORE translator

L. Jerid, F. André, O. Chéron, J.L. Pazat, T. Ernst

N° 2283

Mai 1994

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués

ISSN 0249-6399

HPF to C-PANDORE translator

L. Jerid*, F. André*, O. Chéron*, J.L. Pazat* , T. Ernst†

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet Pampa

Rapport de recherche n° 2283 — Mai 1994 — 36 pages

Abstract: This paper addresses the translation of HPF programs into C-PANDORE ones.

Key-words: Distributed memory parallel computers, compilation, HPF, FORTRAN90, source-to-source translation

(Résumé : tsvp)

*IRISA, Campus de Beaulieu, 35042 Rennes (France)
†GMD-FIRST, Rudower Chaussee 5, D-12489 Berlin (Germany)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Traducteur HPF – C-PANDORE

Résumé : Ce rapport traite de la traduction de programmes HPF en programmes C-PANDORE.

Mots-clé : Machines parallèles à mémoire distribuée, compilation, HPF, FORTRAN90, traduction

Contents

1 Introduction **4**

2 GMD COCKTAIL Toolbox and basis for implementation of translator **4**
2.1 GMD COCKTAIL compiler tools 4
2.1.1 Overall structure 4
2.1.2 Compiler generator 4
2.1.2.1 Scanner generator 4
2.1.2.2 Parser generator 6
2.1.2.3 Generator for Abstract Syntax Trees 6
2.1.2.4 Generator for the transformation of attributed trees 6
2.2 Basis for implementation of translator 6
2.2.1 GMD Fortran90/HPF frontend 6
2.2.2 GMD C code generator 6

3 Adaptation for C-PANDORE **7**
3.1 HPF distribution 7
3.2 C-PANDORE distributed phase 7
3.3 Translation scheme 8

4 Translatable HPF subset **10**
4.1 FORTRAN 90 part of HPF 10
4.2 Specific HPF constructs and directives 11

5 Test suite **11**
5.1 Example: Cholesky factorization 11
5.2 Evaluation 14

6 Future work **15**

7 Appendix: The HPF test suite programs and the C-PANDORE generated code **16**

1 Introduction

In recent years, many projects have focused on the *data distribution* approach to program distributed memory parallel computers [1]. Several language proposals arose from these works, e.g. Vienna Fortran [15] and Fortran D [6]. At IRISA, we concentrated on developing compiling and run-time techniques [13, 14] and did not put the emphasis on the language aspects. For various practical reasons C was chosen as the base sequential language and we extended it with a minimal set of directives, giving birth to the C-PANDORE language [2, 4].

Recently, the High Performance Fortran (HPF) language [5] has been defined as a de-facto standard for expressing data distribution in Fortran programs. It is designed as a set of extensions and modifications to the standard "Fortran 90" [7], and it exploits the parallelism inherent in many scientific applications.

Several programs have already been written in HPF and new benchmarks will be expressed in that language. For this reason, the PANDORE team decided to design and to implement a HPF-to-C-PANDORE translator in order to be able to test the PANDORE compiler with HPF programs. Due to the current set of directives of C-PANDORE, only a subset of HPF may be translated. The purpose of this paper is to present this subset and the way the translator has been built.

The contents of this technical report is organized as follows: section 2 gives an overview of the GMD COCKTAIL toolbox and the source code translator on which the HPF to C-PANDORE translator is based. Section 3 presents the HPF-to-C-PANDORE translation scheme. Section 4 lists the features of the HPF language recognized by the translator. Section 5 presents a list of programs used to test the translator and measurements to compare the automatically C-PANDORE generated codes to hand-written ones.

2 GMD COCKTAIL Toolbox and basis for implementation of translator

Within this section we present a brief description of the compiler generation toolbox and the HPF to sequential C translator on which we based our own translator.

2.1 GMD COCKTAIL compiler tools

2.1.1 Overall structure

Figure 1 shows the co-operation between the different programs which constitute the toolbox. A complete description can be found in [11].

2.1.2 Compiler generator

2.1.2.1 Scanner generator

Rex [10] generates programs to be used in lexical analysis of text. A typical action is the generation of scanners for compilers. *Rex* stands for Regular EXpression tool. It is inspired from LEX. *Rex* processes a specification containing regular expressions to be searched for, and actions written in C or Modula-2 to be executed when expressions are found. *Rex* generates a table-driven scanner consisting of a scanner routine and control tables. The scanner routine implements an automaton and contains a copy of the specified actions.

2.1.2.2 Parser generator

The parser generator *Lalr* [12] has been developed with the aim to combine a powerful specification technique for context-free languages with the generation of highly efficient parsers. The grammars may be written using EBNF constructs. Each grammar rule may be associated with a semantic action consisting of arbitrary statements written in the target language. Whenever a grammar rule is recognized by the parser generator, the associated semantic action is executed. A mechanism for S-attribution (only synthesized attributes) is provided to allow communication between the semantic actions.

2.1.2.3 Generator for Abstract Syntax Trees

Ast [8] is a generator for program modules that defines the structure of abstract syntax trees and provides general tree manipulating procedures. The defined trees may be decorated with attributes of arbitrary types. The structure of the trees is specified by a formalism based on context-free grammars. The generated module includes procedures to construct and to destroy trees, to read and to write trees from (to) files, and to traverse trees in some commonly used manners. *Ast*'s input language is used to define the node types, the subtype relation between node types and attributes of the node types including their data types.

2.1.2.4 Generator for the transformation of attributed trees

Puma [9] is a tool supporting the transformation and manipulation of attributed trees. It is based on pattern-matching, unification, and recursion. *Puma* cooperates with the generator for abstract syntax trees *Ast*. *Puma* adds a concise notation for the analysis and synthesis of trees.

2.2 Basis for implementation of translator

We based our HPF to C-Pandore translator on an existing tool developed by GMD, which translates a subset of HPF into sequential C code.

2.2.1 GMD Fortran90/HPF frontend

At GMD-FIRST, W. Aßmann and P. Enskonatus have developed a Fortran90/HPF frontend. It handles the whole Fortran90 language and can parse all the HPF data distribution annotations. It is the first prototype of the parser of the HPF compilation system which is being developed within the PREPARE European ESPRIT project [3]. This frontend has been implemented with the help of different tools of the COCKTAIL toolbox: *Rex, Lalr, Ast, Puma*. Also, it contains sophisticated hand-written parts to cope with the idiosyncrasies of the source language.

2.2.2 GMD C code generator

The same authors have also developed a C code generator. It can be associated with the Fortran90/HPF parser in order to produce C sequential code from Fortran90/HPF source code. In the early version of the code generator we have used, some Fortran90 features were not handled for instance, array sections. The C code generator is implemented using the *Puma* transformation tool.

3 Adaptation for C-PANDORE

Starting from the GMD C code generator, the adaptation for C-PANDORE consists in the transformation of the existing rules and in the insertion of new rules dedicated to the generation of C-PANDORE distribution information. The implementation of the transformations is based on pattern matching, unification and recursion. Each transformation is mainly implemented by a set of subroutines, each subroutine being composed of rules. A rule consists of a pattern describing a tree fragment and an action. When the current subtree is matched with a pattern, the corresponding action of a rule is executed in order to generate code. In the following sections we present the way data distribution is expressed in HPF and C-PANDORE. We then describe the translation scheme we have implemented.

3.1 HPF distribution

The DISTRIBUTE directive specifies a mapping of data objects to abstract processors in a processor arrangement. It may appear only in the specification part of a scoping unit and be applied to some declared objects, accessible from this scoping unit. Several formats may be used in a DISTRIBUTE directive: BLOCK, CYCLIC, *.
Distributions may be specified independently for each dimension of a multidimensional array.

```
Example:

INTEGER, DIMENSION(N, N) :: A1, A2

!HPF$ PROCESSORS PROC1(P1, P1)
!HPF$ PROCESSORS PROC2(P2)
!HPF$ DISTRIBUTE A1(BLOCK, BLOCK)
!HPF$ DISTRIBUTE A2(CYCLYC, *)
```

The A1 array will be carved up into contiguous rectangular patches, which will be distributed onto a two-dimensional arrangement of abstract processors. The A2 array will have its rows distributed cyclically over a one-dimensional arrangement. The * specifies that A2 is not to be distributed along its second axis; thus an entire row is to be distributed as one object.

3.2 C-PANDORE distributed phase

A C-PANDORE program is a sequential program which calls distributed phases. A distributed phase is spread over the processors of the target distributed memory machine 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 C-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 i^{th} 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.

3.3 Translation scheme

In HPF, directives to express a static distribution, may be used without any restrictions except that these directives should be declared in the specification part of a scoping unit. In the C-Pandore language, all information about data distribution is gathered in one construct: the *distributed phase*. The other constructs are *macros*, functions and *main* program.

To translate a Fortran90/HPF program into C-Pandore, the following cases are to be considered:

- There is no distributed directive in the current program unit.
  The current program unit is:

  `subroutine:` translate the subroutine to a C-Pandore macro construct.

  `function:` if one of the dummy arguments is an array then add the result variable of the function to the list of dummy arguments and translate the function to a C-Pandore macro construct. Otherwise, translate the function to the C-Pandore function construct.

  `main program:` translate the main program to the C-Pandore main program.

- Some data objects are distributed (let L be the list of these objects).
  The current program unit is:

  `subroutine:` add L to the list of dummy arguments of the subroutine and translate it to a C-Pandore distributed phase.

  `function:` add the result variable of the function to the list of dummy arguments and translate the function to a C-Pandore distributed phase.

`main program:` translate the main program to a C-PANDORE distributed phase with L as parameters, and create one C-PANDORE main program. This main program simply calls the distributed phase created above.

`Algorithm:`

```
if( NoDistributedObjects( ListOfDeclaredObjects ) )
{
   if( IsASubroutine( CurrentProgramUnit ) )
      TranslateSubroutineToCPandoreMacro( CurrentProgramUnit )
   else {
     if( IsFunction( CurrentProgramUnit ) ) {
        if( AtLeastOneElementOfTheListIsAnArray( ListOfDummyArgs ) ) {
           ListOfDummyArgs = Concat(ListOfDummyArgs, ResultOfFunction)
           TranslateFunctionToCPandoreMacro( CurrentProgramUnit )
        }
        else TranslateFunctionToCPandoreFunction( CurrentProgramUnit )
     }
     else if( MainProgram( CurrentProgramUnit ) )
             TranslateMainProgramToCPandoreMainProgram( CurrentProgramUnit)
   }
}
else
{
   L = GetListOfDistributedObjects( ListOfDeclaredObjects ) ;
   if( IsASubroutine( CurrentProgramUnit ) )
      TranslateSubroutineToCPandoreDistributedPhase( CurrentProgramUnit, L )
   else {
     if( IsFunction( CurrentProgramUnit ) ) {
        ListOfDummyArgs = Concat(ListOfDummyArgs, ResultOfFunction)
        TranslateFunctionToCPandoreDistributedPhase( CurrentProgramUnit, L )
     }
     else
     if( MainProgram( CurrentProgramUnit ) ) {
        TranslateMainProgramToCPandoreDistributedPhase( CurrentProgramUnit, L )
        CreatedDistributedPhase = GetName ( CurrentProgramUnit )
        CreateCPandoreMainProgram ( L, Call( CreatedDistributedPhase ) )
     }
   }
}
```

Example:

source code:

```
PROGRAM TEST
   REAL, DIMENSION(N, N) :: A, B
!HPF$ PROCESSORS PROCS(32)
```

```
!HPF$ DISTRIBUTE A(CYCLIC, *) ONTO PROCS
   <others declared objects>

   <execution-part>

END PROGRAM TEST
```

target code:

```
dist TEST( float A[N][N] by block (N, 1) map wrapped(1, 0) mode INOUT )
{
{
 <others declared objects>

    <execution-part>
}

main ()
{
  float A[N][N];

  TEST (A);
}
```

4 Translatable HPF subset

This section presents the subset of HPF handled by the HPF to C-PANDORE translator.

4.1 FORTRAN 90 part of HPF

The whole FORTRAN 90 is translated to C-PANDORE except for the following features:

- CHARACTER, COMPLEX, DERIVED types and all operations attached to them,
- Array constructor,
- ALLOCATABLE, POINTER, OPTIONAL and TARGET attributes,
- ALLOCATABLE, POINTER, TARGET, DATA, NAMELIST,
- EQUIVALENCE and COMMON statements,
- Dynamic association,
- ENTRY and CONTAINS statements,
- WHERE and FORALL statements and constructs,
- Renamed entities in a USE statement,

- Input and output statements.

The following restrictions hold:

- Array must be declared with an explicit shape,
- The overlapping between left-hand-side and right-hand-side array sections of an array assignment is only detected in the case where subscript triplets are constant expressions.

4.2 Specific HPF constructs and directives

Two HPF directives are taken into account with the following restrictions:

- PROCESSORS directive: all processor arrangements must have explicit shapes or must be scalar. Processor arrangements declared in the program must have the same shape.
- DISTRIBUTE directive: each dummy argument of a procedure must be explicitly distributed (the use of INHERIT is not allowed).

5 Test suite

The test suite used to experiment the HPF to C-PANDORE translator includes the following programs:

- Cholesky factorization
- LU factorization
- Modified Gram-Schmidt algorithm
- Jacobi algorithm
- Red-Black Successive Over-Relaxation
- Matrix Matrix Product

5.1 Example: Cholesky factorization

source file `CHOLESKY.hpf`

```
!
! Cholesky factorization  - enhanced version
!

! main program
PROGRAM CHOLESKY

   INTEGER, PARAMETER :: N = 512
   REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: Y

   !PRINT *, 'Cholesky Factorization ', N, ' x ', N
```

```
   CALL INIT(Y)
   CALL FACT(Y)
   CALL CHECKSUM(Y)
END PROGRAM CHOLESKY

SUBROUTINE INIT (A)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
   INTEGER I

   DO I=0, N-1
        A(I, I) = N+1  !REAL(N+1, 8)
        A(I,I+1:N-1) = -1.0_8
        A(I+1:N-1,I) = -1.0_8
   END DO
END SUBROUTINE INIT

SUBROUTINE CHECKSUM (A)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
   INTEGER I, J
   REAL(KIND=8) :: SUM

   SUM = 0.0_8
   DO I=0, N-1
        DO J=0, N-1
             SUM = SUM + A(I, J)
        END DO
   END DO
   !PRINT *, 'Checksum = ', SUM
END SUBROUTINE CHECKSUM

SUBROUTINE FACT (A)
   INTEGER, PARAMETER :: N = 512
REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
!HPF$ PROCESSORS PROCS(32)
!HPF$ DISTRIBUTE A(*, CYCLIC) ONTO PROCS

   INTEGER I, J, K
   REAL(KIND=8), DIMENSION(0:N-1) :: COLKA

   DO K=0, N-1
        A(K, K) = SQRT(A(K, K))
        A(K+1:N-1, K) = A(K+1:N-1, K) / A(K, K)
        COLKA(K+1:N-1) = A(K+1:N-1, K)   !broadcast A(K+1:N-1, K)
        DO J=K+1, N-1
             A(J:N-1, J) = A(J:N-1, J) - COLKA(J:N-1) * COLKA(J)
        END DO
   END DO
END SUBROUTINE FACT
```

generated file `CHOLESKY.pa`

```
macro INIT(
double A[512][512])
{
  int I;

  for (I=0;I<=511;I++) {
    A[I][I] = 513;
    {
     int i_1;
        for(i_1=0;i_1<((511-(I+1))+1);i_1++)
           A[i_1+(I+1)][I] = (-1.00000000000000e+00);
    }
    {
     int i_1;
        for(i_1=0;i_1<((511-(I+1))+1);i_1++)
           A[I][i_1+(I+1)] = (-1.00000000000000e+00);
    }
  }
}

macro CHECKSUM(
double A[512][512])
{
  int I;
  int J;
  double SUM;

  SUM = 0.00000000000000e+00;
  for (I=0;I<=511;I++) {
    for (J=0;J<=511;J++) {
      SUM = (SUM+A[J][I]);
    }
  }
}

dist FACT(
double A[512][512] by block (1, 512) map wrapped( 1, 0) mode INOUT)
{
  int I;
  int J;
  int K;
  double COLKA[512];

  for (K=0;K<=511;K++) {
    A[K][K] = sqrt(A[K][K]);
    {
     int i_1;
        for(i_1=0;i_1<((511-(K+1))+1);i_1++)
           A[K][i_1+(K+1)] = (A[K][i_1+(K+1)]/A[K][K]);
    }
```

```
    {
     int i_1;
        for(i_1=0;i_1<((511-(K+1))+1);i_1++)
           COLKA[i_1+(K+1)] = A[K][i_1+(K+1)];
    }
    for (J=(K+1);J<=511;J++) {
      {
       int i_1;
          for(i_1=0;i_1<((511-J)+1);i_1++)
             A[J][i_1+J] = (A[J][i_1+J]-(COLKA[i_1+J]*COLKA[J]));
      }
    }
  }
}

main() {
  double Y[512][512];

  {
     INIT(Y);
  }
  {
     FACT(Y);
  }
  {
     CHECKSUM(Y);
  }
}
```

5.2 Evaluation

In the following table we present the execution times obtained when running the test suite programs. The table compares the automatically generated code, using our translator, with an hand-written code (with the same data distribution).

All the programs are run using a 512x512 double precision matrix, on a 32 nodes iPSC/2.

Table 1:

	Hand-Written (s)	*Automatically Generated (s)*
CHOLESKY	19.55	20.08
MGS	111.41	112.41
JACOBI	0.54	0.51
MATPROD	66.77	67.54
REDBLACK	0.90	0.92

From that we may conclude that the automatically generated code is quite similar to a hand-written one.

Moreover, the overhead of the translator is negligible with respect to the whole compilation time: it takes approximatively 2 seconds to translate 2000 lines of HPF code into C-PANDORE code.

6 Future work

We intend to extend the C-PANDORE language with new directives, mainly concerning the alignment, in order to be able to translate a larger subset of HPF.

Acknowledgement

We would like to thank W. Aßmann and P. Enskonatus from GMD-FIRST who have developed the HPF to sequential C translator on which we based our own translator.

7 Appendix: The HPF test suite programs and the C-PANDORE generated code

source file `MGS.hpf`

```
!
! Modified Gram_Schmid algorithm
!

! main program
PROGRAM MGS
   INTEGER, PARAMETER :: N = 512
   REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: Y

   CALL INIT(Y)
   !PRINT *, 'MGS Matrix ', N, ' x ', N
   CALL VECT(Y)
   CALL CHECKSUM(Y)
END PROGRAM MGS

SUBROUTINE INIT (A)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
   INTEGER I, J

   DO I=0, N-1
        A(I, I) = N+1 !REAL(N+1, 8)
        A(I,I+1:N-1) = -1.0_8
        A(I+1:N-1,I) = -1.0_8
   END DO
END SUBROUTINE INIT

SUBROUTINE CHECKSUM (A)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
   INTEGER I, J
   REAL(KIND=8) :: SUM

   SUM = 0.0_8
   DO I=0, N-1
        DO J=0, N-1
             SUM = SUM + A(I, J)
        END DO
   END DO
   !PRINT *, 'Checksum = ', SUM
END SUBROUTINE CHECKSUM

SUBROUTINE VECT ( V )
   INTEGER, PARAMETER :: N = 512
         REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: V
   REAL(KIND=8), DIMENSION(0:N-1) :: XNORM, SDOT, VC
   INTEGER I, J, K
```

```
!HPF$ PROCESSORS PROCS(32)
!HPF$ DISTRIBUTE V(CYCLIC, *) ONTO PROCS
!HPF$ DISTRIBUTE (CYCLIC) ONTO PROCS :: XNORM, SDOT

! computing vectors
   DO I=0, N-1
        ! normalization
        XNORM(I) = 0.0_8
        DO K=0, N-1
             XNORM(I) = XNORM(I) + V(I, K) * V(I, K)
        END DO
        XNORM(I) = 1.0_8 / SQRT(XNORM(I))
        V(I, :) = V(I, :) * XNORM(I)

        ! global send
        VC(:) = V(I, :)

        ! correction
        DO J=I+1, N-1
             SDOT(J) = 0.0_8 ;
             DO K=0, N-1
                  SDOT(J) = SDOT(J) + VC(K) * V(J, K)
             END DO
             V(J, :) = V(J, :) - SDOT(J) * VC(:)
        END DO
   END DO
END SUBROUTINE VECT
```

generated file **MGS.pa**

```
macro INIT(
double A[512][512])
{
  int I;
  int J;

  for (I=0;I<=511;I++) {
    A[I][I] = 513;
    {
     int i_1;
        for(i_1=0;i_1<((511-(I+1))+1);i_1++)
           A[i_1+(I+1)][I] = (-1.00000000000000e+00);
    }
    {
     int i_1;
        for(i_1=0;i_1<((511-(I+1))+1);i_1++)
           A[I][i_1+(I+1)] = (-1.00000000000000e+00);
    }
  }
}

macro CHECKSUM(
double A[512][512])
{
  int I;
  int J;
  double SUM;

  SUM = 0.00000000000000e+00;
  for (I=0;I<=511;I++) {
    for (J=0;J<=511;J++) {
      SUM = (SUM+A[J][I]);
    }
  }
}

dist VECT(
double V[512][512] by block (512, 1) map wrapped( 1, 0) mode INOUT)

double XNORM[512] by block (1) map wrapped( 0);
double SDOT[512] by block (1) map wrapped( 0);
{
  double VC[512];
  int I;
  int J;
  int K;

  for (I=0;I<=511;I++) {
    XNORM[I] = 0.00000000000000e+00;
```

```
    for (K=0;K<=511;K++) {
      XNORM[I] = (XNORM[I]+(V[K][I]*V[K][I]));
    }
    XNORM[I] = (1.00000000000000e+00/sqrt(XNORM[I]));
    {
     int i_1;
        for(i_1=0;i_1<512;i_1++)
           V[i_1][I] = (V[i_1][I]*XNORM[I]);
    }
    {
     int i_1;
        for(i_1=0;i_1<512;i_1++)
           VC[i_1] = V[i_1][I];
    }
    for (J=(I+1);J<=511;J++) {
      SDOT[J] = 0.00000000000000e+00;
      for (K=0;K<=511;K++) {
        SDOT[J] = (SDOT[J]+(VC[K]*V[K][J]));
      }
      {
       int i_1;
          for(i_1=0;i_1<512;i_1++)
             V[i_1][J] = (V[i_1][J]-(SDOT[J]*VC[i_1]));
      }
    }
  }
}

main() {
  double Y[512][512];

  {
     INIT(Y);
  }
  {
     VECT(Y);
  }
  {
     CHECKSUM(Y);
  }
}
```

source file **JACOBI.hpf**

```
!
! Jacobi algorithm.
!

! main program
PROGRAM JACOBI

   INTEGER, PARAMETER :: N = 512
   INTEGER, PARAMETER :: NBPROCS = 32
   REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: Y


   !PRINT *, 'Jacobi ', N, ' x ', N, ' on ', NBPROCS, ' nodes'
   CALL INIT(Y)
   CALL JACOB(Y)
   CALL CHECKSUM(Y)
END PROGRAM JACOBI

SUBROUTINE INIT (A)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
   INTEGER I, J

     DO I=0, N-1
        DO J=0, N-1
         A(I, J) = I+J
        END DO
     END DO
END SUBROUTINE INIT

SUBROUTINE CHECKSUM (A)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
   INTEGER I, J
   REAL(KIND=8) :: SUM

   SUM = 0.0_8
   DO I=0, N-1
        DO J=0, N-1
             SUM = SUM + A(I, J)
        END DO
   END DO
   !PRINT *, 'Checksum = ', SUM
END SUBROUTINE CHECKSUM

SUBROUTINE JACOB (B)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: B
   REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
```

```
!HPF$ PROCESSORS PROCS(32)
!HPF$ DISTRIBUTE (BLOCK, *) ONTO PROCS :: A, B

   REAL(KIND=8) :: V = 0.5, W = 0.125
   INTEGER, PARAMETER :: NLOOP = 1
   INTEGER I,  K

   DO K=0, NLOOP-1
        DO I=1, N-2
          A(I, 1:N-2) = V*B(I, 1:N-2) + W*(B(I-1,1:N-2)+B(I+1, 1:N-2)+ &
          B(I, 0:N-3)+B(I, 2:N-1))
        END DO
        B(1:N-2, 1:N-2) = A(1:N-2, 1:N-2)
   END DO
END SUBROUTINE JACOB
```

generated file `JACOBI.pa`

```
macro INIT(
double A[512][512])
{
  int I;
  int J;

  for (I=0;I<=511;I++) {
    for (J=0;J<=511;J++) {
      A[J][I] = (I+J);
      ;
    }
    ;
  }
}

macro CHECKSUM(
double A[512][512])
{
  int I;
  int J;
  double SUM;

  SUM = 0.00000000000000e+00;
  for (I=0;I<=511;I++) {
    for (J=0;J<=511;J++) {
      SUM = (SUM+A[J][I]);
    }
  }
}

dist JACOB(
double B[512][512] by block (512, 16) map regular( 1, 0) mode INOUT)

double A[512][512] by block (512, 16) map regular( 1, 0);
{
  double V;
  double W;
  int I;
  int K;

  W = 1.25000000000000e-01;
  V = 5.00000000000000e-01;
  for (K=0;K<=0;K++) {
    for (I=1;I<=510;I++) {
      {
       int i_1;
          for(i_1=0;i_1<510;i_1++)
             A[i_1+1][I] = ((V*B[i_1+1][I]) +
                           (W*(((B[i_1+1][(I-1)]+B[i_1+1][(I+1)]) +
```

```
   B[i_1+0][I])+B[i_1+2][I])));
      }
    }
    {
     int i_1; int i_2;
        for(i_2=0;i_2<510;i_2++)
           for(i_1=0;i_1<510;i_1++)
              B[i_1+1][i_2+1] = A[i_1+1][i_2+1];
    }
  }
}

main() {
  double Y[512][512];

  {
     INIT(Y);
  }
  {
     JACOB(Y);
  }
  {
     CHECKSUM(Y);
  }
}
```

source file **MATPROD.hpf**

```
!
! Matrix Product - enhanced version.
!

! main program
PROGRAM MP
   INTEGER, PARAMETER :: N = 512
   INTEGER, PARAMETER :: NBPROCS = 32
   REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A, B, C

   !PRINT *, 'Matrix Product ', N, ' x ', N, ' on ', NBPROCS, ' nodes'
   CALL INIT(A)
   CALL INIT(B)
   CALL MAT_PROD(A, B, C)
END PROGRAM MP

SUBROUTINE INIT (A)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
   INTEGER I

   DO I=0, N-1
        A(I, I) = N+1  !REAL(N+1, 8)
        A(I, I+1:N-1) = -1.0_8
        A( I+1:N-1, I) = -1.0_8
   END DO
END SUBROUTINE INIT

SUBROUTINE MAT_PROD(A, B, C)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1), INTENT(IN) :: A, B
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1), INTENT(OUT) :: C
!HPF$ PROCESSORS PROCS(32)
!HPF$ DISTRIBUTE A(BLOCK, *) ONTO PROCS
!HPF$ DISTRIBUTE B(*, BLOCK) ONTO PROCS
!HPF$ DISTRIBUTE C(BLOCK, *) ONTO PROCS
   INTEGER I, J, K
   REAL(KIND=8), DIMENSION(0:N-1) :: COLJB

   C = 0.0_8
   DO J=0, N-1
       COLJB = B(:, J) ! broadcast the column j of B
        DO I=0, N-1
             DO K=0, N-1
                  C(I, J) = C(I, J) + A(I, K)*COLJB(K)
             END DO
        END DO
   END DO
END SUBROUTINE MAT_PROD
```

generated file **MATPROD.pa**

```
macro INIT(
double A[512][512])
{
  int I;

  for (I=0;I<=511;I++) {
    A[I][I] = 513;
    {
     int i_1;
        for(i_1=0;i_1<((511-(I+1))+1);i_1++)
           A[i_1+(I+1)][I] = (-1.00000000000000e+00);
    }
    {
     int i_1;
        for(i_1=0;i_1<((511-(I+1))+1);i_1++)
           A[I][i_1+(I+1)] = (-1.00000000000000e+00);
    }
  }
}

dist MAT_PROD(
double A[512][512] by block (512, 16) map regular( 1, 0) mode IN,
double B[512][512] by block (16, 512) map regular( 1, 0) mode IN,
double C[512][512] by block (512, 16) map regular( 1, 0) mode OUT)
{
  int I;
  int J;
  int K;
  double COLJB[512];

  {
   int i_1; int i_2;
      for(i_2=0;i_2<512;i_2++)
         for(i_1=0;i_1<512;i_1++)
            C[i_1][i_2] = 0.00000000000000e+00;
  }
  for (J=0;J<=511;J++) {
    {
     int i_1;
        for(i_1=0;i_1<512;i_1++)
           COLJB[i_1] = B[J][i_1];
    }
    for (I=0;I<=511;I++) {
      for (K=0;K<=511;K++) {
        C[J][I] = (C[J][I]+(A[K][I]*COLJB[K]));
      }
    }
  }
}
```

```
main() {
  double A[512][512];
  double B[512][512];
  double C[512][512];

  {
     INIT(A);
  }
  {
     INIT(B);
  }
  {
     MAT_PROD(A,B,C);
  }
}
```

source file **REDBLACK.hpf**

```
!
! RedBlack SOR - enhanced version.
!

! main program
PROGRAM RB
   INTEGER, PARAMETER :: N = 512
   INTEGER, PARAMETER :: NBPROCS = 32
   REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: Y

   !PRINT *, 'RedBlack ', N, ' x ', N, ' on ', NBPROCS, ' nodes'
   CALL INIT(Y)
   CALL REDBLACK(Y)
   CALL CHECKSUM(Y)
END PROGRAM RB

SUBROUTINE INIT (A)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
   INTEGER I

   DO I=0, N-1
        A(I, I) = N+1  !REAL(N+1, 8)
        A(I, I+1:N-1) = -1.0_8
        A(I+1:N-1, I) = -1.0_8
   END DO
END SUBROUTINE INIT

SUBROUTINE CHECKSUM (A)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
   INTEGER I, J
   REAL(KIND=8) :: SUM

   SUM = 0.0_8
   DO I=0, N-1
        DO J=0, N-1
             SUM = SUM + A(I, J)
        END DO
   END DO
   !PRINT *, 'Checksum = ', SUM
END SUBROUTINE CHECKSUM

SUBROUTINE REDBLACK (A)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
!HPF$ PROCESSORS PROCS(32)
!HPF$ DISTRIBUTE A(BLOCK, *) ONTO PROCS
   REAL(KIND=8) :: W = 1.5
```

```
   INTEGER, PARAMETER :: NLOOP = 1
   INTEGER I, J, K

   DO K=0, NLOOP-1
        DO J=0, (N-1)/2 - 1
             DO I=0, (N-1)/2 - 1
                  A(2*I+1, 2*J+1) = (W/4)*(A(2*I, 2*J+1) + &
                                           A(2*I+2, 2*J+1) + &
                                           A(2*I+1, 2*J) + &
                                           A(2*I+1, 2*J+2)) + &
                                    A(2*I+1, 2*J+1)*(1-W)
             END DO
        END DO

        DO J=0, (N-1)/2
             DO I=0, (N-1)/2
                  A(2*I, 2*j) = (W/4)*(A(2*I-1, 2*J) + &
                                       A(2*I+1, 2*J) + &
                                       A(2*I, 2*J-1) + &
                                       A(2*I, 2*J+1)) + &
                                A(2*I, 2*J)*(1-W)
             END DO
        END DO

        DO J=0, (N-1)/2 - 1
             DO I=0, (N-1)/2
                  A(2*I, 2*j+1) = (W/4)*(A(2*I-1, 2*J+1) + &
                                         A(2*I+1, 2*J+1) + &
                                         A(2*I, 2*J) + &
                                         A(2*I, 2*J+2)) + &
                                  A(2*I, 2*J+1)*(1-W)
             END DO
        END DO

        DO J=0, (N-1)/2
             DO I=0, (N-1)/2 - 1
                  A(2*I+1, 2*J) = (W/4)*(A(2*I, 2*J)+ &
                                         A(2*I+2, 2*J)+ &
                                         A(2*I+1, 2*J-1) + &
                                         A(2*I+1, 2*J+1)) + &
                                  A(2*I+1, 2*J)*(1-W)
             END DO
        END DO
   END DO
END SUBROUTINE REDBLACK
```

generated file **REDBLACK.pa**

```
macro INIT(
        double A[512][512])
{
  int I;

  for (I=0;I<=511;I++) {
    A[I][I] = 513;
    {
     int i_1;
        for(i_1=0;i_1<((511-(I+1))+1);i_1++)
           A[i_1+(I+1)][I] = (-1.00000000000000e+00);
    }
    {
     int i_1;
        for(i_1=0;i_1<((511-(I+1))+1);i_1++)
           A[I][i_1+(I+1)] = (-1.00000000000000e+00);
    }
  }
}

macro CHECKSUM(
        double A[512][512])
{
  int I;
  int J;
  double SUM;

  SUM = 0.00000000000000e+00;
  for (I=0;I<=511;I++) {
    for (J=0;J<=511;J++) {
      SUM = (SUM+A[J][I]);
    }
  }
}

dist REDBLACK(
        double A[512][512] by block (512, 16) map regular( 1, 0) mode INOUT)
{
  double W;
  int I;
  int J;
  int K;
  W = 1.50000000000000e+00;

  for (K=0;K<=0;K++) {
    for (J=0;J<=254;J++) {
      for (I=0;I<=254;I++) {
        A[((2*J)+1)][((2*I)+1)] = (((W/4)*(((A[((2*J)+1)][(2*I)]+
                                  A[((2*J)+1)][((2*I)+2)])+
```

```
                                A[(2*J)][((2*I)+1)])+
                                A[((2*J)+2)][((2*I)+1)]))+
                                (A[((2*J)+1)][((2*I)+1)]*(1-W)));
      }
    }
    for (J=0;J<=255;J++) {
      for (I=0;I<=255;I++) {
        A[(2*J)][(2*I)] = (((W/4)*(((A[(2*J)][((2*I)-1)]+
                          A[(2*J)][((2*I)+1)])+
                          A[((2*J)-1)][(2*I)])+
                          A[((2*J)+1)][(2*I)]))+
                          (A[(2*J)][(2*I)]*(1-W)));
      }
    }
    for (J=0;J<=254;J++) {
      for (I=0;I<=255;I++) {
        A[((2*J)+1)][(2*I)] = (((W/4)*(((A[((2*J)+1)][((2*I)-1)]+
                              A[((2*J)+1)][((2*I)+1)])+
                              A[(2*J)][(2*I)])+
                              A[((2*J)+2)][(2*I)]))+
                              (A[((2*J)+1)][(2*I)]*(1-W)));
      }
    }
    for (J=0;J<=255;J++) {
      for (I=0;I<=254;I++) {
        A[(2*J)][((2*I)+1)] = (((W/4)*(((A[(2*J)][(2*I)]+
                              A[(2*J)][((2*I)+2)])+
                              A[((2*J)-1)][((2*I)+1)])+
                              A[((2*J)+1)][((2*I)+1)]))+
                              (A[(2*J)][((2*I)+1)]*(1-W)));
      }
    }
  }
}

main() {
  double Y[512][512];

  {
     INIT(Y);
  }
  {
     REDBLACK(Y);
  }
  {
     CHECKSUM(Y);
  }
}
```

source file **LU.hpf**

```
!  LU Factorization


PROGRAM MAIN

INTEGER, PARAMETER :: N = 512
REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A


   !PRINT *, 'LU Factorization ', N, 'x', N
   CALL INIT (A)
   CALL FACTORISATION (A)
   CALL CHECKSUM (A)

END PROGRAM MAIN

SUBROUTINE INIT (A)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
   INTEGER I, J

   DO I=0, N-1
        A(I, I) = N+1  !REAL(N+1, 8)
        A(I,I+1:N-1) = -1.0_8
        A(I+1:N-1,I) = -1.0_8
   END DO
END SUBROUTINE


SUBROUTINE CHECKSUM (A)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1) :: A
   INTEGER I, J
   REAL(KIND=8) :: SUM

   SUM = 0.0_8
   DO I=0, N-1
        DO J=0, N-1
             SUM = SUM + A(I, J)
        END DO
   END DO
   !PRINT *, 'Checksum = ', SUM
END SUBROUTINE CHECKSUM

SUBROUTINE FACTORISATION (A)
   INTEGER, PARAMETER :: N = 512
        REAL(KIND=8), DIMENSION(0:N-1,0:N-1), INTENT(INOUT) :: A
!HPF$ PROCESSORS PROCS(32)
!HPF$ DISTRIBUTE A(CYCLIC, *) ONTO PROCS
```

```
   INTEGER I, J, K
   REAL(KIND=8), DIMENSION(0:N-1) :: LINE_K
   REAL(KIND=8) :: CORNER


   DO K=0, N-2
        CORNER = A(K, K)
        A(K+1:N-1, K) = A(K+1:N-1, K) / CORNER
        LINE_K(K+1:N-1) = A(K, K+1:N-1)
        DO I=K+1, N-1
             A(I, K+1:N-1) = A(I, K+1:N-1) - A(I,K)*LINE_K(K+1:N-1)
        END DO
   END DO
END SUBROUTINE FACTORISATION
```

generated file `LU.pa`

```
macro INIT(
double A[512][512])
{
  int I;
  int J;

  for (I=0;I<=511;I++) {
    A[I][I] = 513;
    {
     int i_1;
        for(i_1=0;i_1<((511-(I+1))+1);i_1++)
           A[i_1+(I+1)][I] = (-1.00000000000000e+00);
    }
    {
     int i_1;
        for(i_1=0;i_1<((511-(I+1))+1);i_1++)
           A[I][i_1+(I+1)] = (-1.00000000000000e+00);
    }
  }
}

macro CHECKSUM(
double A[512][512])
{
  int I;
  int J;
  double SUM;

  SUM = 0.00000000000000e+00;
  for (I=0;I<=511;I++) {
    for (J=0;J<=511;J++) {
      SUM = (SUM+A[J][I]);
    }
  }
}

dist FACTORISATION(
double A[512][512] by block (512, 1) map wrapped( 1, 0) mode IN)
{
  int I;
  int J;
  int K;
  double LINE_K[512];
  double CORNER;
  int i_1;

  for (K=0;K<=510;K++) {
    CORNER = A[K][K];
    {
```

```
        for(i_1=0;i_1<((511-(K+1))+1);i_1++)
           A[K][i_1+(K+1)] = (A[K][i_1+(K+1)]/CORNER);
    }
    {
     int i_1;
        for(i_1=0;i_1<((511-(K+1))+1);i_1++)
           LINE_K[i_1+(K+1)] = A[i_1+(K+1)][K];
    }
    for (I=(K+1);I<=511;I++) {
      {
       int i_1;
          for(i_1=0;i_1<((511-(K+1))+1);i_1++)
             A[i_1+(K+1)][I] = (A[i_1+(K+1)][I]-(A[K][I]*LINE_K[i_1+(K+1)]));
      }
    }
  }
}

main() {
  double A[512][512];

  {
     INIT(A);
  }
  {
     FACTORISATION(A);
  }
  {
     CHECKSUM(A);
  }
}
```

References

[1] Françoise André, Olivier Chéron, Marc Le Fur, Yves Mahéo, and Jean-Louis Pazat. Programmation des machines à mémoire distribuée par distribution des données : langages et compilateurs. *Techniques et Sciences Informatiques*, 12(5):563–596, November 1993.

[2] Françoise André, Olivier Chéron, and Jean-Louis Pazat. Compiling Sequential Programs for Distributed Memory Parallel Computers with Pandore II. In Jack J. Dongarra and Bernard Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 293–308, Elsevier Science Publishers B.V., 1993. *Également disponible en rapport de recherche IRISA no. 651.*

[3] Arthur Veen and Martijn de Lange. Overview of the Prepare Project. In *Fourth Workshop on Compilers for Parallel Computers*, Delft, Holland, December 1993.

[4] Olivier Chéron. *Pandore II : un compilateur dirigé par la distribution des données.* PhD thesis, IFSIC/Université de Rennes I, July 1993.

[5] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Technical Report Version 1.0, Rice University, May 1993.

[6] Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, and Chau-Wen Tseng. *An Overview of the Fortran D Programming System.* Technical Report TR91121, Center for Research on Parallel Computation, Rice University, March 1991.

[7] International Standard ISO/IEC. *Information technology – Programming languages – Fortran.* ISO/IEC, July 1991.

[8] Josef Grosch. *Ast - A Generator for Abstract Syntax Trees.* Technical Report Compiler Generation Report No. 15, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1992.

[9] Josef Grosch. *Puma - A Generator for the Transformations of Attributed Trees.* Technical Report Compiler Generation Report No. 26, GMD Forschungsstelle an der Universität Karlsruhe, Nov. 1991.

[10] Josef Grosch. *Rex - A Scanner Generator.* Technical Report Compiler Generation Report No. 5, GMD Forschungsstelle an der Universität Karlsruhe, July 1992.

[11] Josef Grosch and Helmut Emmelmann. *A Tool Box for Compiler Construction.* Technical Report Compiler Generation Report No. 20, GMD Forschungsstelle an der Universität Karlsruhe, Jan. 1990.

[12] Josef Grosch and Bertram Vielsack. *The Parser Generators Lalr and Ell.* Technical Report Compiler Generation Report No. 8, GMD Forschungsstelle an der Universität Karlsruhe, July 1992.

[13] M. Le Fur, J.L. Pazat, and F. André. *Static Domain Analysis for Compiling Commutative Loop Nests.* Technical Report 757, IRISA, September 1993.

[14] Yves Mahéo and Jean-Louis Pazat. *Distributed Array Management for HPF Compilers.* Technical Report, IRISA, 1993.

[15] Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and Andreas Schwald. *Vienna Fortran - A Language Specification.* Technical Report, ICASE, NASA, March 1992.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399