Evaluation of automatic parallelization strategies for HPF compilers

Pierre Boulet, Thomas Brandes

Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Evaluation of Automatic Parallelization Strategies for HPF Compilers

Pierre Boulet
Thomas Brandes

November 1995

Research Report N^{O} 95-44

Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Evaluation of Automatic Parallelization Strategies for HPF Compilers

Pierre Boulet
Thomas Brandes

November 1995

Abstract

In the data parallel programming style the user usually specifies the data parallelism explicitly so that the compiler can generate efficient code without enhanced analysis techniques.

In some situations it is not possible to specify the parallelism explicitly or this might be not very convenient. This is especially true for loop nests with data dependences between the data of distributed dimensions.

In the case of uniform loop nests there are scheduling, mapping and partitioning techniques available. Some different strategies have been considered and evaluated with existing High Performance Fortran compilation systems.

This paper gives some experimental results about the performance and the benefits of the different techniques and optimizations. The results are intended to direct the future development of data parallel compilers.

Keywords: data parallelism, High Performance Fortran, loop nests, automatic parallelization, compilation, optimization

Résumé

Dans le style de programmation data-parallèle, l'utilisateur spécifie habituellement le data-parallélisme explicitement de façon à permettre au compilateur de générer du code efficace sans techniques d'analyse avancées.

Dans certaines situations, il n'est pas possible de spécifier le parallélisme explicitement ou ce n'est pas très pratique. C'est particulièrement vrai dans le cas des nids de boucles avec des dépendances entre les données des dimensions réparties.

Dans le cas des nids de boucles uniformes, des techniques d'ordonnancement, d'allocation et de partitionnement sont disponibles. Des stratégies différentes ont été considérées et évaluées avec des systèmes de compilation d'High Performance Fortran existants.

Ce rapport donne des résultats expérimentaux de performance et quantifie les bénéfices apportés par les différentes techniques et optimisations. Ces résultats ont pour but d'orienter le développement futur des compilateurs data-parallèles.

Mots-clés: data-parallélisme, High Performance Fortran, nids de boucles parallélisation automatique, compilation, optimisation

Evaluation of Automatic Parallelization Strategies for HPF Compilers

Pierre Boulet
LIP, ENS Lyon *†
Pierre.Boulet@lip.ens-lyon.fr

Thomas Brandes
SCAI, GMD ‡
Thomas.Brandes@gmd.de

1 Introduction

High Performance Fortran (HPF) is a language definition [10, 12] that allows to use the data parallel programming style as a high level parallel programming model within Fortran applications. The data parallelism can be specified by array operations, by the `FORALL` statement and construct, by new library procedures and through the `INDEPENDENT` directive.

Nevertheless many applications contain also implicit parallelism that should be detected and utilized, e.g. some algorithms have inherent, input-independent conflicts between computation and communication. In the example below both loops within the loop nest have data dependences, and neither can be specified as a parallel loop.

```
PARAMETER (N=...)
REAL, DIMENSION (N,N)  :: A
...
DO I = 2, N-1
  DO J = 2, N-1
    A(I,J) = (A(I,J-1)+A(I-1,J)+A(I,J+1)+A(I+1,J))*0.25
  END DO
END DO
```

Automatic parallelization of such loop nests has been studied by many people and some tools for automatic parallelization have been written: SUIF [8], PIPS [17], the Omega Library [16], LooPo [9] and PAF [18] among others.

Research compilers for data parallel Fortran applications have been developed in the Superb [19], Kali [13], Fortran 77D [11], ADAPT [14], and in the Fortran 90D/HPF [4] projects. The first available commercial HPF compilers were the xHPF compiler from Applied Parallel Research [1], the `pghpf` compiler from the Portland Group [15] and the DEC Fortran 90 compiler. Other compilers have been announced or are now available. Though some of these compilers are able to identify parallel loops, none of them is currently dealing with the hyperplane method for loop nest parallelization.

*Laboratoire de l'Informatique du Parallélisme, CNRS URA 1398, Ecole Normale Supérieure de Lyon, 46, Allée d'Italie, 69364 Lyon Cedex 07, France
†Supported by the ReMaP project jointly operated by CNRS and INRIA
‡Institute for Algorithms and Scientific Computing, German National Research Center for Information Technology, Schloss Birlinghoven, P.O. Box 1319, 53754 St. Augustin, Germany

For this evaluation the Bouclettes parallelizer [2] and the ADAPTOR compilation system [5] have been coupled with each other in such a way that Bouclettes can generate HPF codes that could be compiled efficiently with the ADAPTOR tool. While ADAPTOR uses most advanced optimization techniques, the particularities of Bouclettes in regards of the other tools are the employed methodologies and the output language.

2 Description of the Tools

2.1 Overview of the Bouclettes System

The Bouclettes loop parallelizer applies to perfectly nested loops where the loop bounds are affine functions of the surrounding loop indices and of some parameters. All arrays must be fully dimensional and data access functions must be translations.

After the parallelization, the loop nest is rewritten as an outermost sequential loop and inner parallel loops. Furthermore, some HPF directives are included in the final code to specify the array distribution and alignment.

The Bouclettes system is organized as a succession of stages:

1. The input program is analyzed and translated into an internal representation.
2. Then the data dependences are analyzed by a simple custom dependence analyzer to get the exact data dependences.
3. From there, a schedule is computed. It is a function that associates a time of execution to each instance (iteration) of a statement. The user has the choice between two scheduling functions:

   the linear schedule is a linear function that associates a time t to an iteration point $\vec{i}$ ($\vec{i} = (i, j, k)$ if the loop nest is three dimensional) as follows:

   $$t = \left\lfloor \frac{p}{q} \pi . \vec{i} \right\rfloor$$

   where p, q are integers and π is a vector of integers of dimension the depth d of the loop nest with all components prime with each other.

   the shifted linear schedule is an extension of the linear schedule where each statement of the loop nest body has its own scheduling function. All these functions share the same linear part and some (possibly different) shifting constant are added for each statement. The time t for statement k is computed as follows:

   $$t = \left\lfloor \frac{p}{q} \pi . \vec{i} + \frac{c_k}{q} \right\rfloor$$

   where p, q, c_k are whole numbers and π is a vector of whole numbers of dimension d with all components prime with each other.

   The computation of these schedules is done by techniques which guarantee that the result is optimal in the considered class of schedules. Here "optimal" means that the total latency is minimized.

4. The data arrays are then mapped in a compatible way with the schedule. Based on the computation of the so called "communication graph", a structure that represents all the communications that can occur in the given loop nest, a projection M and some shifting constants are computed. The base idea is to project the arrays (and the computations) on a virtual processor grid of dimension $d-1$. Then, the arrays and the computations are aligned (by the shifting constants) to suppress some computations.

5. Finally, the HPF code with explicit parallel loops and a data distribution is generated following the previously computed transformation.

   Many problems appear here. In all the cases, the code generation involves rewriting the loop nest according to a unimodular transformation. This rewriting technique is described in [6] and involves calls to the PIP [7] (Parallel Integer Programming) software. A complete description of the rewriting process can be found in [3].

   The code generation basically produces a sequential loop, representing the iteration over the time given by the schedule surrounding $d-1$ parallel (`INDEPENDENT`) loops scanning the active processors. The arrays are distributed and aligned by HPF directives to respect the mapping previously computed.

   Some complications are induced in many cases:

   the owner computes rule: when the mapping does not satisfy to this rule, some temporary arrays are used to simulate it.

   the projection direction: the expressivity of the `DISTRIBUTE` HPF directive is restricted to projections along axes of the iteration domain. When the mapping projects the data in an other direction, the data are redistributed. This redistribution is done by copying the arrays in new temporary arrays —which are projected along one axis of the domain—, computing the loop nest with these new arrays and finally copying back the results into the original arrays.

   the rationals and time shifting constants: these parameters complicate a lot the generated code, and we would need some control parallelism to fully express the parallelism obtained by this kind of schedule.

   the affine array access functions: a noticeable thing is that the redistribution implemented in the case of non axis parallel projections induces a nearly inverse transformation on array access functions. They are then translations. So it is easier to optimize the produced code. In the following study, we will compare the redistributed programs and the non redistributed ones.

   different output: Bouclettes can generate its output in different HPF subsets. The one that respects the most the theoretical transformations uses `INDEPENDENT` directives to tag the parallel loops. Bouclettes can also generate real world HPF with `FORALL` parallel loops. This kind of HPF code indicates less parallelism but is compilable by current HPF compilers. Furthermore, the best data distribution would distribute the arrays in a `BLOCK-CYCLIC` manner. Bouclettes can generate any data distribution and as ADAPTOR only implements `BLOCK` distributions in its current release, we will only consider `BLOCK` distributions in the following.

2.2 Overview of the ADAPTOR System

ADAPTOR (Automatic Data Parallelism Translator) is a system developed at GMD for compiling data parallel programs to equivalent message passing programs. The system supports most features of the data parallel languages that are used in the context of Fortran: Connection Machine Fortran (CMF) and High Performance Fortran (HPF) [5].

ADAPTOR allows the use of the data parallel programming model for MIMD machines already for over two years while commercial HPF compilers are just coming up. During the development of the system, more attention has been paid to the reliability of the system and to the correctness and efficiency of the generated programs than to language completeness.

ADAPTOR has been made available as public domain software. The comments of many users helped to improve the functionality and stability of the translation tool.

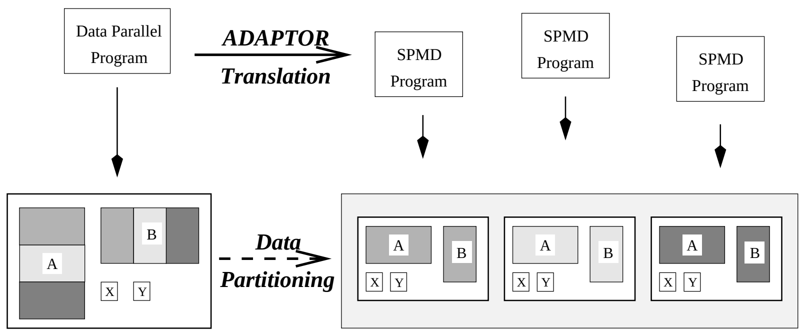

Figure 1: MIMD partitioning with ADAPTOR

By means of a source-to-source transformation, ADAPTOR translates the data parallel program to an equivalent SPMD program (single program, multiple data) that runs on all available nodes. The essential idea of the translation is to distribute the arrays of the source program onto the node processors where the parallel loops and array operations are restricted to the local part owned by the processor (see Figure 1). Communication statements for exchanging non local data is generated automatically. The control flow and statements with scalar code are replicated on all nodes.

With the latest ADAPTOR release (3.1) a lot of optimizations have been implemented.

Regarding the target language of the generated SPMD program, ADAPTOR is very flexible. It can not only generate Fortran 77 or Fortran 90 programs with message passing, but also Fortran 77 with some additional features (e.g. dynamic arrays, array operations).

Beside the translation system, a runtime system called DALIB (distributed array library) has been developed that will be linked with the generated message passing program (see Figure 2). It realizes functions for global reductions, transposition, gather and scatter operations, circular shifting, replication and redistribution of distributed and local arrays. Timing and tracing facilities as well as a random number generator are also part of this library. As the runtime system is available on most parallel systems, the generated message passing programs will run on all these machines.

The evaluation of ADAPTOR for real applications and the development of optimization techniques have been funded by the Esprit project PPPE (Portable Parallel Programming Environment).

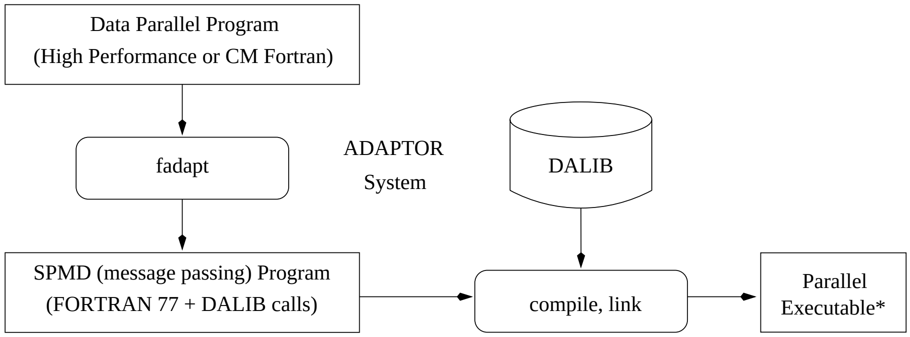

Figure 2: Overview of the ADAPTOR tool

3 Results

In this section it will be presented which efficiency can be achieved with current compiler technology. All results are measured on an IBM SP 2 with thin nodes, AIX version 3.2, PGHPF compiler version 1.3, ADAPTOR version 3.1, and the XL Fortran compiler version 3.2.

We will present two examples, the Gauss-Seidel relaxation and a non-real world example we have called “Mtest”.

3.1 Gauss-Seidel Relaxation

3.1.1 The Serial Code

Here is the serial code that was used here:

```
      PARAMETER (N=...)
      REAL, DIMENSION (N,N)  :: A
      ...
      DO I = 2, N-1
        DO J = 2, N-1
          A(I,J) = (A(I,J-1)+A(I-1,J)+A(I,J+1)+A(I+1,J))*0.25
        END DO
      END DO
```

Starting with this serial code, the PGI HPF compiler and the ADAPTOR system were not able to detect any parallelism. Both compilers generate SPMD code like the following one:

```
      DO J = 2, N-1
        DO I = 2, N-1
c         get local copies of non-local values A(I,J-1) and A(I,J+1)
          IF (HAVE I A(I,J)) then
             A(I,J) = (A(I,J-1)+A(I-1,J)+A(I,J+1)+A(I+1,J))*0.25
          END IF
        END DO
      END DO
```

While the Portland compiler broadcasts the non-local values, the ADAPTOR compiler will exchange the values between neighbor processors but with more overhead to compute ownerships (see table 1).

The dramatical overhead caused by the sending/receiving of single non-local values and fixing the ownerships dominates the computation in the program completely.

N = 256	1 node	2 nodes	4 nodes	8 nodes
PGI HPF	7.6 s	4.8 s	3.8 s	2.5 s
ADAPTOR	4.8 s	4.9 s	4.8 s	4.6 s

Table 1: Results of the serial code

3.1.2 The Bouclettes Generated Code

For the given loop nest the Bouclettes system generates the following code:

```
      PARAMETER (N=...)
      REAL, DIMENSION (N,N)  :: A
!HPF$ DISTRIBUTE A(*,BLOCK)
      ...
      DO T = 4, 2*N-2
         FORALL (J=MAX(2,T-N+1):MIN(N-1,T-2))
     &      A(T-J,J) = (A(T-J,J-1)+A(T-J-1,J)+A(T-J,J+1)+A(T-J+1,J))*0.25
      END DO
```

The PGI HPF compiler was not able to take advantage of the data parallelism in the `FORALL loop` and therefore the SPMD code is still using serial loops with nearly the same execution times as before. The ADAPTOR system takes advantage of the parallelism within the `FORALL` statement. Table 2 shows the results.

ADAPTOR	1 node	2 nodes	4 nodes	8 nodes
N = 256	0.06 s	0.14 s	0.15 s	0.16 s
N = 512	0.11 s	0.35 s	0.40 s	0.42 s
N = 1024	0.78 s	1.36 s	1.37 s	1.35 s
N = 2048	3.55 s	3.66 s	4.06 s	4.84 s
N = 4096	13.42 s	20.05 s	18.34 s	15.56 s

Table 2: Results for the code generated by Bouclettes

The affine array access functions in the generated code have the effect that the HPF compiler cannot optimize the code and exchanges more data in each time step.

3.1.3 Bouclettes Generated Code with Redistributions

By a redistribution that is also generated by the Bouclettes tool, it can be guaranteed that the array access functions become translations. In this case the HPF compilers can optimize the communication.

```
      PARAMETER (N=...)
      REAL, DIMENSION (N,N)  :: A
      REAL, DIMENSION (N,2:2*N)  :: A1
!HPF$ DISTRIBUTE A1(*,BLOCK)
      ...
      FORALL (I=1:N,J=1:N) A1(I,I+J) = A(I,J)
      ...
      DO T = 4, 2*N-2
         FORALL (2=MAX(2,T-N+1):MIN(N-1,T-2))
```

```
&      A1(I,T) = (A1(I,T-1)+A1(I-1,T-1)+A1(I,T+1)+A1(I+1,T+1))*0.25
 END DO
 ...
 FORALL (I=1:N,J=1:N) A(I,J) = A1(I,I+J)
```

With the ADAPTOR system, the parallel execution of the loop nest on the redistributed array gives the results shown in table 3. If compared to the results without the redistribution, there is now a rather good scalability. Also the PGI HPF compiler now generates efficient code by taking advantage of the data parallelism in the loop and by using efficient communication.

ADAPTOR	1 node	2 nodes	4 nodes	8 nodes
N = 256	0.05 s	0.11 s	0.12 s	0.12 s
N = 512	0.15 s	0.24 s	0.26 s	0.28 s
N = 1024	0.91 s	0.80 s	0.63 s	0.60 s
N = 2048	3.48 s	2.71 s	1.79 s	1.43 s
N = 4096	mem !	9.42 s	5.31 s	3.67 s

Table 3: Results for the code with redistributed array

Nevertheless, the redistribution itself and additional memory for the new array has also to be taken into account. The PGI HPF compiler has serialized the whole redistribution and the total execution times are like the serial ones. With ADAPTOR the redistribution is parallelized and does not dominate the computation time, but is still not very efficient. This is due to the fact that the affine indexes do now appear in the redistribution.

3.2 Results for Mtest

In this section the advantages of shifted linear schedules will be presented.

3.2.1 The Serial Code

For our measurements we used a three-dimensional code (Mtest). The computation time and the size of the arrays are of the order $O(N^3)$.

```
PARAMETER (N=...)
REAL, DIMENSION (N,N,N) :: A, B, C, D
...
DO i = 2, n-1
  DO j = 7, n-5
    DO k = 3, n-6
      a(i,j,k) = (b(i,j-6,k-1)+d(i-1,j+3,k+1))
      b(i+1,j-1,k) = c(i+2,j+5,k+2)
      c(i+3,j-1,k-2) = a(i,j-2,k)
      d(i,j-1,k) = a(i,j-1,k+6)
    END DO
  END DO
END DO
```

Table 4 shows the execution times for the serial code (using the XL Fortran compiler).

xlf	1 node
N = 50	0.21 s
N = 75	0.85 s
N = 100	2.76 s
N = 150	14.12 s

Table 4: Results for the serial code Mtest

3.2.2 Linear Schedule with Redistribution

For the given loop nest the Bouclettes system generates the following code (linear schedule). A redistribution is necessary in any case because the projection matrix M computed for the mapping does not correspond to a projection along one axis of the domain.

$$M = \begin{pmatrix} 1 & 1 & -5 \\ 0 & 0 & 1 \end{pmatrix}.$$

This comes from the linear schedule obtained: $(\frac{3}{4}, \frac{1}{2}, \frac{-9}{4})$.

Here is the code generated by Bouclettes for this schedule:

```
      REAL, DIMENSION (14*n-13,7*n-6,n) :: ROT_1_a, ROT_1_b
      REAL, DIMENSION (14*n-13,7*n-6,n) :: ROT_1_c, ROT_1_d

!HPF$ TEMPLATE BCLT_0_template(14*n+143,7*n+174,n+6)
!HPF$ DISTRIBUTE BCLT_0_template(*,BLOCK,BLOCK)
!HPF$ ALIGN ROT_1_a(i1,i2,i3) WITH BCLT_0_template(i1+156,i2,i3+6)
!HPF$ ALIGN ROT_1_b(i1,i2,i3) WITH BCLT_0_template(i1+76,i2+87,i3+2)
!HPF$ ALIGN ROT_1_c(i1,i2,i3) WITH BCLT_0_template(i1+72,i2+104,i3+4)
!HPF$ ALIGN ROT_1_d(i1,i2,i3) WITH BCLT_0_template(i1,i2+180,i3)
      ...
      FORALL (J0 = 1:n,J1 = 1:n,J2 = 1:n)
         ROT_1_a(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2) = a(J0,J1,J2)
         ROT_1_b(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2) = b(J0,J1,J2)
         ROT_1_c(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2) = c(J0,J1,J2)
         ROT_1_d(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2) = d(J0,J1,J2)
      END FORALL

     DO J0 = -9*n+74, 5*n-46

        FORALL (J1 = ceiling(max(-2*n+(J0+43)/3.0,
     &                           -n+(J0+9)/2.0,
     &                           (-7*n+5*J0+23)/9.0)):
     &              floor(min((n+J0-23)/3.0,
     &                        (J0-5)/2.0,
     &                        (5*J0-19)/9.0)),
     &         J2 = ceiling(max(3.0,(J0+7)/6.0-J1/2.0,
     &                              -n+J0-2*J1+3.0)):
     &             floor(min(n-6.0,(n+J0-5)/6.0-J1/2.0,
     &                              J0-2*J1-2.0)))

          ROT_1_a(9*n+J0-4,5*n+J1-1,J2) =
     &       ROT_1_b(9*n+J0-7,5*n+J1-2,J2-1)+
     &       ROT_1_d(9*n+J0-10,5*n+J1-4,J2+1)

          ROT_1_b(9*n+J0-3,5*n+J1-1,J2) =
```

```
&            ROT_1_c(9*n+J0-6,5*n+J1-4,J2+2)

         ROT_1_c(9*n+J0+21,5*n+J1+11,J2-2) =
&           ROT_1_a(9*n+J0-8,5*n+J1-3,J2)

         ROT_1_d(9*n+J0-6,5*n+J1-2,J2) =
&           ROT_1_a(9*n+J0-60,5*n+J1-32,J2+6)
    END FORALL
 END DO

 FORALL (J0 = 1:n,J1 = 1:n,J2 = 1:n)
    a(J0,J1,J2) = ROT_1_a(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2)
    b(J0,J1,J2) = ROT_1_b(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2)
    c(J0,J1,J2) = ROT_1_c(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2)
    d(J0,J1,J2) = ROT_1_d(9*n+3*J0+2*J1-9*J2-4,5*n+J0+J1-5*J2-1,J2)
 END FORALL
```

ADAPTOR and PGI HPF compiler were not able to take advantage of the data parallelism. Table 5 shows the results for a very small problem size (N=30).

The following problems could be identified with the generated code:

- The generated code needs very big arrays that need about 100 times (7×14) more memory than the arrays of the serial version. For this reason the code runs only for very small problem sizes.
- The redistribution is done very inefficiently due to the affine indexes (e.g. `3*J0+2*J1-9*J2`) that prevents the compilers from generating efficient code for the communication.
- In the computational part only the innermost of the two parallel loops is really parallelized. As the innermost loop index `J2` depends on the outermost one `J1`, the iteration space is not rectangular and the compiler did not generate efficient code.
- The compiler did not benefit from overlap areas.

ADAPTOR	1 node	2 nodes	4 nodes	8 nodes
copy in	13.9 s	11.6 s	12.4 s	11.5 s
copy out	13.5 s	11.3 s	12.1 s	11.1 s
computation	3.1 s	4.3 s	4.5 s	5.1 s
total	30.5 s	27.2 s	29.0 s	27.7 s
PGI HPF	1 node	2 nodes	4 nodes	8 nodes
copy in	7.4 s	5.6 s	7.7 s	5.5 s
copy out	7.2 s	5.2 s	7.9 s	6.3 s
computation	3.8 s	4.4 s	8.3 s	26.2 s
total	18.4 s	15.2 s	23.9 s	38.1 s

Table 5: Results for the code generated by Bouclettes (linear schedule)

If the two-dimensional distribution of the template is changed to a one-dimensional distribution,

```
!HPF$ DISTRIBUTE BCLT_0_template(*,*,BLOCK)
```

the generated code is more efficient as now overlap areas are utilized and less communication is generated.

3.2.3 Shifted Linear Schedule

This is the code with the linear shifted schedule without redistribution.

```
!HPF$ TEMPLATE BCLT_0_template(n+3,n+6,n)
!HPF$ DISTRIBUTE BCLT_0_template(BLOCK,BLOCK,*)
!HPF$ ALIGN a(i1,i2,i3) WITH BCLT_0_template(i1,i2+5,i3)
!HPF$ ALIGN b(i1,i2,i3) WITH BCLT_0_template(i1+2,i2,i3)
!HPF$ ALIGN c(i1,i2,i3) WITH BCLT_0_template(i1+3,i2+6,i3)
!HPF$ ALIGN d(i1,i2,i3) WITH BCLT_0_template(i1,i2+5,i3)
      ....
      DO NT = -n+6, -n+8
        FORALL (KT = max(-n+6,NT-2):NT-2,J1 = 7:n-5,J2 = 2:n-3)
     &     a(J2,J1,-KT) = b(J2,J1-6,-KT-1)+d(J2-1,J1+3,-KT+1)
        FORALL (KT = max(-n+6,NT):NT,J1 = 7:n-5,J2 = 2:n-3)
     &     b(J2+1,J1-1,-KT) = c(J2+2,J1+5,-KT+2)
        FORALL (KT = max(-n+6,NT-3):NT-3,J1 = 7:n-5,J2 = 2:n-3)
     &     c(J2+3,J1-1,-KT-2) = a(J2,J1-2,-KT)
        FORALL (KT = max(-n+6,NT-2):NT-2,J1 = 7:n-5,J2 = 2:n-3)
     &     d(J2,J1-1,-KT) = a(J2,J1-1,-KT+6)
      END DO

      DO NT = -n+9, -4
        KT = NT-2
        FORALL (J1 = 7:n-5,J2 = 2:n-3)
     &     a(J2,J1,-KT) = b(J2,J1-6,-KT-1)+d(J2-1,J1+3,-KT+1)
        KT = NT
        FORALL (J1 = 7:n-5,J2 = 2:n-3)
     &     b(J2+1,J1-1,-KT) = c(J2+2,J1+5,-KT+2)
        KT = NT-3
        FORALL (J1 = 7:n-5,J2 = 2:n-3)
     &     c(J2+3,J1-1,-KT-2) = a(J2,J1-2,-KT)
        KT = NT-2
        FORALL (J1 = 7:n-5,J2 = 2:n-3)
     &     d(J2,J1-1,-KT) = a(J2,J1-1,-KT+6)
      END DO

      DO NT = -3, 0
        FORALL (KT = NT-2:min(-3,NT-2),J1 = 7:n-5,J2 = 2:n-3)
     &     a(J2,J1,-KT) = b(J2,J1-6,-KT-1)+d(J2-1,J1+3,-KT+1)
        FORALL (KT = NT:min(-3,NT),J1 = 7:n-5,J2 = 2:n-3)
     &     b(J2+1,J1-1,-KT) = c(J2+2,J1+5,-KT+2)
        FORALL (KT = NT-3:min(-3,NT-3),J1 = 7:n-5,J2 = 2:n-3)
     &     c(J2+3,J1-1,-KT-2) = a(J2,J1-2,-KT)
        FORALL (KT = NT-2:min(-3,NT-2),J1 = 7:n-5,J2 = 2:n-3)
     &     d(J2,J1-1,-KT) = a(J2,J1-1,-KT+6)
      END DO
```

ADAPTOR could generate very efficient code (see table 6). It also scales very well. The PGI HPF compiler was not able to generate this efficient code.

One very nice result is that the code running on one node is faster than the serial counterpart. This is given by the fact the innermost loops are running over the first index and so the cache is better used (stride 1 for array access). It should be noted that it is nearly pure chance. This comes form the linear part of the schedule which is $(0, 0, -1)$ and the alignment matrix which is $M = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$.

ADAPTOR	1 node	2 nodes	4 nodes	8 nodes
50	0.07 s	0.13 s	0.11 s	0.16 s
75	0.23 s	0.32 s	0.20 s	0.24 s
100	0.63 s	0.65 s	0.43 s	0.40 s
150	2.24 s	2.23 s	1.17 s	0.87 s
200	mem !	5.27 s	2.84 s	1.69 s
250	mem !	mem !	5.41 s	3.05 s
PGI HPF	1 node	2 nodes	4 nodes	8 nodes
50	2.83 s	4.63 s	4.39 s	4.14 s
75	8.85 s	6.23 s	5.79 s	4.66 s
100	21.38 s	13.48 s	12.36 s	9.31 s
150	88.70 s	46.18 s	42.73 s	29.70 s
200	mem !	118.82 s	123.67 s	64.24 s

Table 6: Results for the code with shifted linear schedule

In any case, the advantage of the shifted linear schedule is given by the fact that no affine indexes are generated. It is not true in general. But, as the shifted linear schedule is more general than the linear schedule (when all constants are null, the shifted linear schedule becomes a linear schedule), the probability to have a "simple" schedule is larger with the shifted linear schedule. Indeed, the fastest integer linear part one can have is a vector with one 1 (or -1) and the remaining components 0. And this kind of vector gives a very simple projection matrix which projects along one axis, thus generating no affine indexes and making the redistribution needless.

3.2.4 Code with Shifted Linear Schedule and Redistribution

The generated redistribution reverses only the first dimension.

```
FORALL (J0 = 1:n,J1 = 1:n,J2 = 1:n)
    ROT_1_a(n-J2+1,J1,J0) = a(J0,J1,J2)
    ROT_1_b(n-J2+1,J1,J0) = b(J0,J1,J2)
    ROT_1_c(n-J2+1,J1,J0) = c(J0,J1,J2)
    ROT_1_d(n-J2+1,J1,J0) = d(J0,J1,J2)
END FORALL
```

In fact the generated loops make only the task of the compiler more difficult.

```
 DO NT = -n+9, -4
   KT = NT-2
   FORALL (J1 = 7:n-5,J2 = 2:n-3)
&      ROT_1_a(n+KT+1,J1,J2) =
&          ROT_1_b(n+KT+2,J1-6,J2)+ROT_1_d(n+KT,J1+3,J2-1)
   KT = NT
   FORALL (J1 = 7:n-5,J2 = 2:n-3)
&      ROT_1_b(n+KT+1,J1-1,J2+1) = ROT_1_c(n+KT-1,J1+5,J2+2)
   KT = NT-3
   FORALL (J1 = 7:n-5,J2 = 2:n-3)
&      ROT_1_c(n+KT+3,J1-1,J2+3) = ROT_1_a(n+KT+1,J1-2,J2)
   KT = NT-2
   FORALL (J1 = 7:n-5,J2 = 2:n-3)
&      ROT_1_d(n+KT+1,J1-1,J2) = ROT_1_a(n+KT-5,J1-1,J2)
 END DO
```

ADAPTOR fails to recognize that it can use overlap areas. Therefore it creates full copies of the arrays which requires some more copying of data. The computation times are about a factor of 3 longer than in the previous case. Also the redistribution itself needs some time.

ADAPTOR	1 node	2 nodes	4 nodes	8 nodes
copy in	1.80 s	1.32 s	0.80 s	0.40 s
copy out	1.45 s	1.56 s	1.08 s	0.59 s
computation	3.04 s	1.59 s	1.11 s	0.47 s

Table 7: Shifted linear schedule with redistribution (N=100)

It should be mentioned that the redistribution for this code is much faster than the redistribution for the code with the linear schedule.

What we can see from this example is that the theoretical superiority of shifted linear schedules can be verified in practise. But it should be mentioned that there is no general rule here. Indeed, as the compiler may optimize more one version or the other, the opposite situation may arise on another example. Though, we believe that shifted linear schedules are often better than linear ones.

4 Conclusions

The results verify the benefits of automatic loop parallelization. Current technology allows to take advantage of this parallelism in HPF compilers.

We have also been able to identify where current HPF compilers have to be improved. The main problem is to identify the communication between two time steps and to find closed formulas for their computation. Especially the support of affine indexes might improve the performance of Bouclettes generated code dramatically.

Redistributions make the task of the compiler easier but a big overhead is still present and additional problems arise, e.g. overhead for redistributions, additional memory. Furthermore, redistributions would be necessary for all aligned arrays, otherwise there will be problems in other places in the parallel program.

Currently, the implemented methods in Bouclettes are not considering a pipelined execution of the loops to combine messages of different time steps. Comparison of these methods with pipelined execution show that this should also be considered for future versions.

References

[1] FORGE 90. xHPF 1.0 Automatic Parallelizer for High Performance Fortran on Distributed Memory Systems - User's Guide. Technical report, Applied Parallel Research, Inc., April 1993.

[2] P. Boulet. The bouclettes loops parallelizer. Research Report 95-40, Laboratoire de l'informatique du parallélisme, École Normale Supérieure de Lyon, France, Nov 1995.

[3] P. Boulet and M. Dion. Code generation in bouclettes. Research report, Laboratoire de l'informatique du parallélisme, École Normale Supérieure de Lyon, France, Nov 1995.

[4] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation, and Performance Results. Technical Report, Syracuse Center for Computational Science, April 1993.

[5] Th. Brandes and F. Zimmermann. ADAPTOR - A Transformation Tool for HPF Programs. In K.M. Decker and R.M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 91–96. Birkhäuser, April 1994.

[6] Jean-François Collard, Paul Feautrier, and Tanguy Risset. Construction of do loops from systems of affine constraints. *Parallel Processing Letters*, 1994. to appear.

[7] Paul Feautrier and Nadia Tawbi. Résolution de systèmes d'inéquations linéaires; mode d'emploi du logiciel PIP. Technical Report 90-2, Institut Blaise Pascal, Laboratoire MASI (Paris), January 1990.

[8] Stanford Compiler Group. Suif compiler system. World Wide Web document, URL: `http://suif.stanford.edu/suif/suif.html`.

[9] The group of Pr. Lengauer. The loopo project. World Wide Web document, URL: `http://brahms.fmi.uni-passau.de/cl/loopo/index.html`.

[10] High Perforamnce Fortran Forum. High Performance Fortran Language Specification. Final Version 1.0, Department of Computer Science, Rice University, May 1993.

[11] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluating Compiler Optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21:27–45, April 1994.

[12] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[13] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

[14] J. Merlin. ADAPTing Fortran 90 Array Programs for Distributed Memory Architectures. In *Proc. 1st International Conference of the Austrian Center for Parallel Computation*, Salzburg, October 1991.

[15] PGHPF. Reference Manual, User's Guide. Technical report, The Portland Group, Inc., November 1994.

[16] William Pugh and the Omega Team. The omega project. World Wide Web document, URL: `http://www.cs.umd.edu/projects/omega/index.html`.

[17] PIPS Team. Pips (interprocedural parallelizer for scientific programs). World Wide Web document, URL: `http://www.cri.ensmp.fr/~pips/index.html`.

[18] PRiSM SCPDP Team. Systematic construction of parallel and distributed programs. World Wide Web document, URL: `http://www.prism.uvsq.fr/english/parallel/paf/autom_us.html`.

[19] H. Zima, H. Bast, and M. Gerndt. Superb: A Tool for Semi-Automatic SIMD/MIMD Parallelizatin. *Parallel Computing*, January 1988.