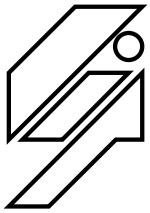
The bouclettes loop parallelizer.

Pierre Boulet



Laboratoire de l'Informatique du Parallélisme

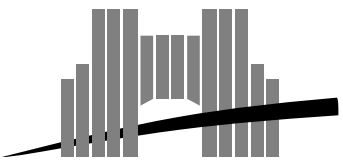
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

The Bouclettes *loop parallelizer*

Pierre BOULET

November 1995

Research Report N^{O} 95-40



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

The Bouclettes loop parallelizer

Pierre BOULET

November 1995

Abstract

Bouclettes is a source to source loop nest parallelizer. It takes as input Fortran uniform, perfectly nested loops and gives as output an HPF (High Performance Fortran) program with data distribution and parallel (`$HPF! INDEPENDENT`) loops. This paper presents the tool and the underlying parallelization methodology.

Keywords: automatic parallelization, loop nest, HPF, compiler

Résumé

Bouclettes est un paralléliseur source à source de nids de boucles. Il prend en entrée des boucles Fortran uniformes et parfaitement imbriquées et retourne en sortie un programme HPF (High Performance Fortran) avec une distribution des données et des boucles parallèles (`$HPF! INDEPENDENT`). Ce papier présente l'outil et les méthodes employées.

Mots-clés: parallélisation automatique, nid de boucles, HPF, compilation

The Bouclettes loop parallelizer

Pierre BOULET <Pierre.Boulet@lip.ens-lyon.fr>

1 Introduction

In the data parallel programming paradigme, the user usually specifies the parallelism explicitely. In some situations, it is difficult to find the parallelism and to express it. Some automatic parallelization tools have been written to address this problem. Bouclettes is such a tool. It takes as input some sort of Fortran loop nest and gives as output an HPF (High Performance Fortran) with explicit parallel constructs. This tool is presented here.

1.1 What is Bouclettes?

Bouclettes has been written to validate some scheduling and mapping techniques based on extensions of the hyperplane method. These techniques are briefly sketched in section 2. The goal pursued when building Bouclettes was to have a completely automatic parallelization tool. This goal has been reached and the input of the user is only required to choose the parallelization methodology which he wants to apply.

We have chosen HPF as the output language because we believe it can become a standard for parallel programming (and thus be widely used). Furthermore, data parallelism is a programmation paradigm that provides a simple way of describing data distributions and of managing the communications induced by the computations. It thus relieves the programmer (or the parallelization tool) from generating the low-level communications inside the parallel program.

This paper is organized as follows: after the introduction, we present the different transformation stages. We then present a detailed parallelization example in section 3. The tool and its possibilities are described in section 4 and we finally conclude in section 5.

1.2 Related work

Automatic parallelization has been studied by many people and some tools for automatic parallelization have been written: SUIF [10], at Stanford University, California, PIPS [15] at the École Nationale Supérieure des Mines de Paris, France, the Omega Library [14] at the University of Maryland, Maryland, LooPo [11] at the University of Passau, Germany, and PAF [16] at the University of Versailles, France, among others.

The particularities of Bouclettes in regards of these other tools are the employed methodologies (see section 2) and the output language (HPF).

2 Data analysis and parallelism extraction

The parallelization process can be decomposed into several inter-dependent tasks. See figure 1.

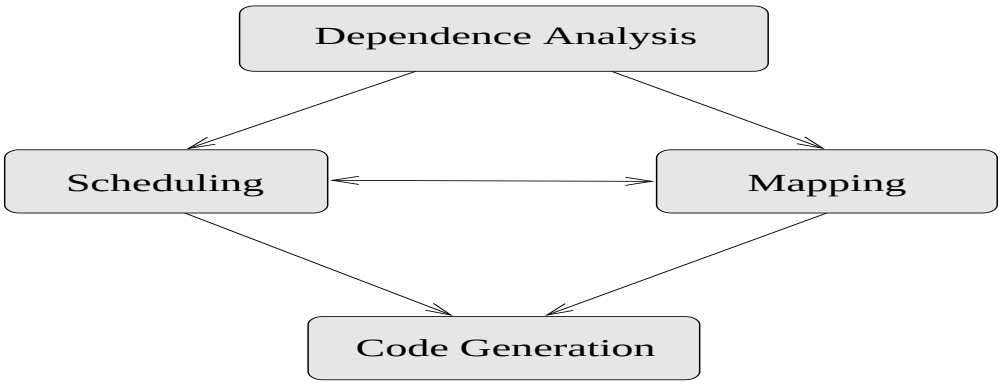


Figure 1: The parallelization stages

The dependence analysis consists in building a graph representing the constraints on the execution order of the instances of the statements. The scheduling uses the dependences to build a function that associates an execution time to an instance of a statement. The mapping stage maps the data arrays and the instances of the statements to a virtual set of processors. The two previous stages (the scheduling and the mapping) are inter-dependent: we want that the global transformation of the original loop nest respects the data dependences. The last stage is the code generation. We generate here code with parallel loops (`INDEPENDENT` loops) and a data allocation (`DISTRIBUTE` and `ALIGN` directives).

The Bouclettes system is organized as a succession of stages:

1. the input program is analyzed and translated into an internal representation
2. this representation is used to compute the data dependences; in our case, data dependences are uniform, so a simple custom dependence analyzer is powerful enough to determine the exact data dependences
3. from these data dependences, a linear or shifted linear schedule is computed
4. the schedule and the internal representation are used to compute a mapping compatible with the schedule.
5. finally, the HPF code is generated following the previously computed transformation.

2.1 Dependence analysis

The dependence analysis is quite simple in the restricted context we have here. It basically consists in finding all the data dependences between the inner state-

ments. The three kinds of dependences (direct, anti and output dependences) can be computed in the same way: the dependence vectors are differences between two data acess functions that address the same array; and reciprocally, all the differences between two data acess functions that address the same array are dependence vectors.

2.2 Scheduling

Darte and Robert have presented techniques to compute schedules for a given uniform loop nest [3, 5]. These techniques are part of the theoretical basis of Bouclettes [1].

Currently, the user has the choice between linear scheduling and shifted linear scheduling.

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:

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

where p, q are integers and π is a vector of integers whose components are relatively prime 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_k for statement k is computed as follows:

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

where p, q, c_k are integers and π is a vector of integers 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.

2.3 Mapping

Darte and Robert have presented a technique to build a mapping of data and computation on a virtual processor grid [4]. It is this technique that is used in Bouclettes [2].

Based on the computation of the so called "communication graph", a structure that represents all the communications that can occur in the given loop

[1] Some of the scheduling techniques have been extended by Darte and Vivien [6] from linear scheduling to multi-dimensional schedules. These techniques will be included in a future version of Bouclettes.

[2] Dion and Robert have extended this technique to affine loop nests [7]. This will be implemented in a future version of Bouclettes.

nest, a projection M and some alignment constants are computed. The basic 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 alignment constants) to suppress some computations.

More precisely, M, the projection matrix, is a $(d-1) \times d$ full-rank matrix of integers and the constants α_x are vectors of integers and of dimension $d-1$. Each array or statement x is then associated with an allocation function defined by:

$$\text{alloc}_x(\vec{i}) = M\vec{i} + \alpha_x$$

As the considered loop nests are uniform, choosing a different matrix for different arrays or statements would not improve the mapping. The schedule has to be taken into account to choose the matrix M. Effectively, the transformed loop nest iteration domain will be the image of the initial iteration domain by the transformation:

$$\vec{i} \mapsto \begin{bmatrix} \pi \\ M \end{bmatrix} \vec{i}$$

It is mandatory to have this iteration domain mapped onto $\mathbf{N}^d$, because otherwise we would need rationally indexed processors. As the choice of M does not have a high impact on the number of communications that remain, $\left[\begin{smallmatrix} \pi \\ M \end{smallmatrix}\right]$ is just computed as the unimodular completion of vector π.

Once M has been computed, the alignment constants are determined in order to minimize the number of communications. Here the user can choose if he wants to respect the owner computes rule (as in HPF) or not. If he chooses not to respect this rule, some temporary arrays may be generated in the next stage to take this into account.

2.4 The code generation

Many problems appear here. In all cases, the code generation involves rewriting the loop nest according to a unimodular transformation. This rewriting technique is described in [2] and involves calls to the PIP [8] (Parallel Integer Programming) software. A complete description of the rewriting process can be found in [1].

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 verify this rule, some temporary arrays are used to simulate it.

the projection direction: the expressiveness of the `DISTRIBUTE` HPF directive is restricted to projections along axes of the iteration domain. When the mapping projects the data in another direction, we redistribute the data. 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 the 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 distribution strategy: The best data distribution would be to distribute the arrays in a block-cyclic manner with the size of the blocks depending on the target machine. This would partition the data such as to equilibrate the computation load and in the same time allow the compiler to group some communications so that they take less time. Bouclettes can generate any distribution. As current HPF compilers only understand block distributions, to be able to test the output of Bouclettes, we generate block distributions.

3 A detailed example

We will study here the parallelization process on an example.

3.1 The input program

The example that we consider (see program 1) is a two dimensional loop nest with two inner statements. This is not a real world code but has been designed to show the parallelization process.

Program 1 Input program

```
      parameter (n=100)

      integer i,j
      real a(n,n)
      real b(n,n)

      do i= 2, n-5
         do j= 6, n-3
c   Statement 1
            a(i,j)=a(i,j-5)+b(i-1,j+3)-a(i+5,j-4)
c   Statement 2
            b(i+1,j-2)=a(i,j-1)
         enddo
      enddo
```

There are four data dependences which are:

From statement	To statement	Dependence vector
2	1	$[2,-5]$
1	2	$[0,1]$
1	1	$[5,-4]$
1	1	$[0,5]$

The first line means that the data item produced by the instance $[i, j]$ of statement 2 is used by the instance $[i + 2, j - 5]$ statement 1.

3.2 Linear scheduling without redistribution

The optimal linear scheduling vector is $[3, 1]$. The projection matrix is $[1, 0]$ and the alignment constants are:

array	shift	statement	shift
a	0	1	0
b	1	2	0

We can easily check that the owner computes rule is respected here. Actually forcing to respect this rule here gives another equivalent mapping. As the owner computes rule is respected, we will not see any temporary arrays in the produced code to enforce this rule. As the projection matrix is $[1, 0]$, the projection of the two-dimensional arrays is done on the first dimension following the second direction. Hence the redistribution is not necessary.

The resulting code is program 2[3].

Program 2 HPF program with linear schedule and no redistribution

```
      PROGRAM boucle

      INTEGER P1
      INTEGER T
      PARAMETER (n = 100)
      REAL a(n,n)
      REAL b(n,n)

!HPF$ TEMPLATE BCLT_0_template(n+1,n+3)
!HPF$ DISTRIBUTE BCLT_0_template(BLOCK,*)
!HPF$ ALIGN a(i1,i2) WITH BCLT_0_template(i1,i2+3)
!HPF$ ALIGN b(i1,i2) WITH BCLT_0_template(i1+1,i2)
      DO T = 12, 4*n-18
!HPF$ INDEPENDENT
         DO P1 = ceiling(max((-n+T)/3.0+1,2)), floor(min((T)/3.0-2,n-5))
            a(P1,T-3*P1) = (a(P1,T-3*P1-5)+(b(P1-1,T-3*P1+3)-a(P1+5,T-3*P1-4)))
            b(P1+1,T-3*P1-2) = a(P1,T-3*P1-1)
         END DO
      END DO
      END
```

Following the declarations, there are the distribution and alignment directives. They are generated from the mapping projection and the shifting constants. The `DISTRIBUTE` directive uses a `BLOCK` strategy to map the template on the processors. As mentionned before, any strategy could be used here and a block-cyclic approach with a block size depending on the target machine would probably be a better solution.

[3] The variables have been renamed for improved readability

The loop nest consists in a sequential loop (index T) surrounding a parallel loop (index P1). T and P1 are obtained from the initial loop indices as:

$$\begin{pmatrix} \texttt{T} \\ \texttt{P1} \end{pmatrix} = \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \texttt{i} \\ \texttt{j} \end{pmatrix}$$

3.3 Linear scheduling with redistribution

The user can choose to enforce a redistribution. In this case, the data arrays are copied into temporary arrays for which the resulting loop nest may be simpler to analyze by an HPF compiler. This is because when doing this redistribution, the complicated array acces functions are moved out of the main loop nest to the surrounding FORALL loops that realize the redistribution. Thus, the array access functions become translations that are better optimized by HPF compilers. The resulting HPF program is program 3.

The FORALL statements express the redistribution before and after the computation of the transformed loop nest using the temporary arrays. Note that the array access functions are translations in this case and are more complicated without the redistribution.

3.4 Shifted linear scheduling case

We study here shifted linear scheduling without redistribution. It is possible to redistribute the arrays as in the previous section but it would not show anything new and would only complicate the code here.

The shifted linear scheduling functions[4] are:

$$\begin{cases} \text{schedule}_1(I) = \left\lfloor \frac{1}{5}([7,1].I) \right\rfloor \\ \text{schedule}_2(I) = \left\lfloor \frac{1}{5}([7,1].I + 4) \right\rfloor \end{cases}$$

The mapping of the arrays (and the computations) is the same as the one obtained for linear scheduling and the resulting code is decomposed in three stages:

1. The initial stage (see program 4) is limited to one unit of time (T = 4) and we can see the max function in the lower bound of the loops over the virtual time VT. This function ensures that the computations start at the right time considering the time shifting constants. Each statement is inside a loop nest of depth two: the VT index iterates over the instances of the statement that are scheduled at the same time (here T = 4), and the P1 index iterates over the (virtual) processors.

2. The steady-state stage (see program 5) is the main stage when there is no time boundary problem and every thing is regular. Once again we have the two parallel loop nests inside the sequential loop over the time.

3. The final stage (see program 6) matches the initial stage to deal with the end of the computations in respect of the shifting constants.

[4] $\text{schedule}_i(I)$ is the scheduling function of statement i

Program 3 HPF program with linear schedule and redistribution

```
      PROGRAM boucle

      INTEGER I1
      INTEGER I0
      INTEGER P1
      INTEGER T
      PARAMETER (n = 100)
      REAL a(n,n)
      REAL b(n,n)
      REAL ROTa(4*n-3,n)
      REAL ROTb(4*n-3,n)

!HPF$ TEMPLATE BCLT_0_template(4*n,n+10)
!HPF$ DISTRIBUTE BCLT_0_template(*,BLOCK)
!HPF$ ALIGN ROTa(i1,i2) WITH BCLT_0_template(i1+3,i2)
!HPF$ ALIGN ROTb(i1,i2) WITH BCLT_0_template(i1,i2+10)
      FORALL (I0 = 1:4*n-3,I1 = 1:n)
         ROTa(I0,I1) = 0
      END FORALL
      FORALL (I0 = 1:4*n-3,I1 = 1:n)
         ROTb(I0,I1) = 0
      END FORALL
      FORALL (I0 = 1:n,I1 = 1:n)
         ROTa(3*I0+I1-3,I0) = a(I0,I1)
      END FORALL
      FORALL (I0 = 1:n,I1 = 1:n)
         ROTb(3*I0+I1-3,I0) = b(I0,I1)
      END FORALL
      DO T = 12, 4*n-18
!HPF$ INDEPENDENT
         DO P1 = ceiling(max((-n+T)/3.0+1,2)), floor(min((T)/3.0-2,n-5))
            ROTa(T-3,P1) = (ROTa(T-8,P1)+(ROTb(T-3,P1-1)-ROTa(T+8,P1+5)))
            ROTb(T-2,P1+1) = ROTa(T-4,P1)
         END DO
      END DO
      FORALL (I0 = 1:n,I1 = 1:n)
         a(I0,I1) = ROTa(3*I0+I1-3,I0)
      END FORALL
      FORALL (I0 = 1:n,I1 = 1:n)
         b(I0,I1) = ROTb(3*I0+I1-3,I0)
      END FORALL
      END
```

Program 4 The initialization stage

```
      PROGRAM boucle

      INTEGER T
      INTEGER VT
      INTEGER P1
      PARAMETER (n = 100)
      REAL a(n,n)
      REAL b(n,n)

!HPF$ TEMPLATE BCLT_0_template(n+1,n+7)
!HPF$ DISTRIBUTE BCLT_0_template(BLOCK,*)
!HPF$ ALIGN a(i1,i2) WITH BCLT_0_template(i1,i2+7)
!HPF$ ALIGN b(i1,i2) WITH BCLT_0_template(i1+1,i2)
      T = 4
!HPF$ INDEPENDENT
      DO VT = max(20,5*T), 5*T+4
!HPF$ INDEPENDENT
         DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
            a(P1,VT-7*P1) = (a(P1,VT-7*P1-5)+
     & (b(P1-1,VT-7*P1+3)-a(P1+5,VT-7*P1-4)))
         END DO
      END DO
!HPF$ INDEPENDENT
      DO VT = max(20,5*T-4), 5*T
!HPF$ INDEPENDENT
         DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
            b(P1+1,VT-7*P1-2) = a(P1,VT-7*P1-1)
         END DO
      END DO
```

Program 5 The steady-state stage

```
      DO T = 5, (floor((8*n-38)/5.0)-1)
!HPF$ INDEPENDENT
         DO VT = 5*T, 5*T+4
!HPF$ INDEPENDENT
            DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
               a(P1,VT-7*P1) = (a(P1,VT-7*P1-5)+
     & (b(P1-1,VT-7*P1+3)-a(P1+5,VT-7*P1-4)))
            END DO
         END DO
!HPF$ INDEPENDENT
         DO VT = 5*T-4, 5*T
!HPF$ INDEPENDENT
            DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
               b(P1+1,VT-7*P1-2) = a(P1,VT-7*P1-1)
            END DO
         END DO
      END DO
```

Program 6 The final stage

```
      DO T = floor((8*n-38)/5.0), floor((8*n-34)/5.0)
!HPF$ INDEPENDENT
         DO VT = 5*T, min(8*n-38,5*T+4)
!HPF$ INDEPENDENT
            DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
               a(P1,VT-7*P1) = (a(P1,VT-7*P1-5)+
     & (b(P1-1,VT-7*P1+3)-a(P1+5,VT-7*P1-4)))
            END DO
         END DO
!HPF$ INDEPENDENT
         DO VT = 5*T-4, min(8*n-38,5*T)
!HPF$ INDEPENDENT
            DO P1 = ceiling(max((-n+VT+3)/7.0,2)), floor(min((VT-6)/7.0,n-5))
               b(P1+1,VT-7*P1-2) = a(P1,VT-7*P1-1)
            END DO
         END DO
      END DO
      END
```

4 The tool and its possibilities

4.1 The implementation

Bouclettes has been written using the Caml Light [5] language [12, 13]. We have choosen Caml Light because it is makes it very easy to handle complex data structures such as abstract syntax trees or symbolic expressions, and to do symbolic computations. As lexical and syntactical analyzers are also easy to write in Caml Light, it is very well suited to write compilers. To make Bouclettes, we have developped some utility modules such as modules to do rationnal computation, symbolic general and affine expression manipulation, or matrix computations and an interface to the PIP software to do the parametric integer programming. We have interfaced Caml Light with the C programs that compute the schedules using files. And the graphical user interface has been written using the Caml Tk library allowing a nicely integrated program.

4.2 The user interface

Bouclettes comes in two forms: a command line program that accept options and does all the processing in one step and a graphical user interface that allows the user to see the different stages of the transformation one at a time and interactively set or change the options.

The available options are the following:

- the choice of the schedule type: linear or shifted linear
- the choice to enforce or not the owner computes rule
- the choice to redistribute the data even when it is not necessary

[5] Caml Light is an implementation of ML made by INRIA

There is one more option in Bouclettes: the user can choose the output language for testing purposes. All the outputs other than the standard HPF one include the initial filling of the arrays and the computation of the sum of all the elements of the arrays after the computation of the transformed loop nest. The different output formats are:

- the input program in Fortran 77.
- the output program in Fortran 77. All the HPF directives are removed and the `INDEPENDENT` and `FORALL` loops are translated into seqential `DO` loops, thus allowing to check if the result computed by the output program is the same as the one computed by the input program
- the output program in a subset of HPF that is understood by current HPF compilers. Indeed, all the current HPF compilers only implement a subset of the HPF 1 standard [9]. In particular, the `INDEPENDENT` loops and `FORALL` constructs are translated into `FORALL` statements. This is also the reason why a block distribution has been chosen.

5 Conclusion

We have presented here a loop nest parallelization tool, Bouclettes that has been realized at the LIP of the ENS Lyon. This tool parallelizes some kind of Fortran loop nests into HPF. It uses some scheduling techniques derived from the hyperplane method, namely the linear schedule and shifted linear scheduling, and some mapping techniques to distribute the data and reduce the communications. The code is then finally rewritten in HPF.

More information about Bouclettes (installation guide, reference manual, papers presenting the theoretical methodologies) can be found at the URL

`http://www.ens-lyon.fr/~pboulet/bclt/bouclettes.html`

Acknowledgment

We would like to thank all the people who have contributed to the writing of this tool: Michèle Dion, Tanguy Risset and Frédéric Vivien.

References

[1] Pierre Boulet and Michèle Dion. The code generation in bouclettes. Technical report, Laboratoire de l'Informatique du Parallélisme, 1995.

[2] Jean-François Collard, Paul Feautrier, and Tanguy Risset. Construction of do loops from systems of affine constraints. Technical Report 93-15, Laboratoire de l'Informatique du Parallélisme, may 1993.

[3] Alain Darte, Leonid Khachiyan, and Yves Robert. Linear scheduling is nearly optimal. *Parallel Processing Letters*, 1(2):73–81, 1991.

[4] Alain Darte and Yves Robert. The alignment problem for perfect uniform loop nest: Np-completeness and heuristics. In J.J. Dongarra and B. Tourancheau eds, editors, *Environments and Tools for Parallel Scientific Computing II*, SIAM Press, pages 33–42, 1994.

[5] Alain Darte and Yves Robert. Constructive methods for scheduling uniform loop nests. *IEEE Trans. Parallel Distributed Systems*, 5(8):814–822, 1994.

[6] Alain Darte and Frédéric Vivien. Automatic parallelization based on multi-dimensional scheduling. Technical Report 94-24, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, France, September 1994.

[7] Michèle Dion and Yves Robert. Mapping affine loop nests: New results. In Bob Hertzberger and Guiseppe Serazzi, editors, *High-Performance Computing and Networking, International Conference and Exhibition*, volume LCNS 919, pages 184–189. Springer-Verlag, 1995. Extended version available as Technical Report 94-30, LIP, ENS Lyon (anonymous ftp to lip.ens-lyon.fr).

[8] 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.

[9] Hich Performance Fortran Forum. High performance fortran language specification. Technical report, Rice University, January 1993.

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

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

[12] Xavier Leroy and Pierre Weis. *Manuel de Référence du Langage Caml*. Inter Editions, 1994.

[13] projet Cristal. The caml language. World Wide Web document, URL:
`http://pauillac.inria.fr/caml/`.

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

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

[16] 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.