Scheduling Reductions

Xavier Redon and Paul Feautrier
e-mail : Xavier.Redon@prism.uvsq.fr,
Paul.Feautrier@prism.uvsq.fr.
Laboratoire PRiSM,
Université de Versailles-Saint-Quentin,
45, Avenue des États-Unis,
78035 Versailles Cedex (France).

Abstract

In order to detect more parallelism in scientific programs, one may extract parallelism relative to reductions. This paper presents such a method which schedules programs with explicit computations of reductions. We describe the way the reductions are expressed in our input language (which is in fact the output language of the reductions detector presented in [RF93]). We also give a brief summary of scheduling techniques. In order to simplify the scheduling we suppose that the target parallel computer has an infinite number of processors with infinite fan-in. We show that a schedule computed with this model can be adapted to work on real parallel machines. Then we present a scheduling method based on the algorithms from [Fea92a, Fea92b] which works in presence of reductions. This method is applied on an example. Lastly, we show that side-effects of reductions scheduling are the simplification of the scheduling process and the improvement of the computed schedules.

1 Introduction

The basic algorithms of linear algebra and matrix computation fall into two broad classes. In the first one, the result is of the same size or bigger than the data. This is the case, for instance, for vector operations. In the second class, the result is much smaller – typically only one value – than the data, hence the name *reduction* which has been coined by Iverson [Ive62].

There is hope of finding parallel implementations of algorithms in the first class: we may expect that each element of the result is at least partially independent of other parts. There is no such possibility for the second class, and more sophisticated methods are necessary here.

One possibility is to use the fact that, in many cases, the operations involved in computing reductions have specific algebraic properties. One often has to combine an ordered set of values by an associative binary operator. The algorithm may be implemented as a computation on a binary tree, with significant parallelism if the tree is balanced. This procedure may be used to advantage on multiprocessors. Other techniques are necessary on pipeline computers.

ICS 94 - 7/94 Manchester, U. K.
© 1994 ACM 0-89791-665-4/94/0007..$3.50

In usual programming languages, reductions are implemented as sequential loops. Ordinary vectorizing or parallelizing compilers do not take into account operator semantics, and hence cannot find the hidden parallelism in these special loops. Some commercial compilers recognize common reductions as idioms, and then translate them into calls to a run-time library. This approach is severely limited because there are many fancy ways of implementing a reduction, especially when it is interspersed with other calculations. Recent research (see [RF93] and the references therein) has focused on a two-steps approach. The source program is put in normal form, which is then subjected to pattern recognition. The size of the pattern base is inversely proportional to the power of the normalization algorithm. Section 2 gives a short presentation of the reduction detector of [RF93].

Alternatively, a language in which reductions can be expressed, like ALPHA or CRYSTAL ([Mau89] and [MCL88]), may be used. Whatever the situation, one still has the problem of generating the parallel code for a source program in which reductions may be interspersed in a complicated way with more ordinary calculations. Callahan [Cal91] presents a method to parallelize recurrences by a direct code transformation. The principal advantage of this method is its rapidity. But since there is no attempt to simplify (i.e. normalize) the initial program, the final parallel code contains an important overhead. It is interesting to note that executing our normalization phase before Callahan's code transformation will lead to better parallel programs. In fact, we chose a different way for producing parallel code. Indeed, many researchers in the field of parallel program construction have found that a very powerful way of assessing the parallelism of a program uses a *schedule*, i.e. a closed form function which gives the date at which each operation in the program may be executed. Among other information, a comparison of the length of the schedule to the length of the sequential calculation gives the main degree of parallelism or *grain* of the algorithm. Section 3 is a brief review of recent results on scheduling for computers with an infinite number of processors.

To extend this work to programs with reductions, we must first decide on which machine model the schedule is to be used. We expect a schedule to give target architecture independent information on the source program. However, the machine model should be realistic enough to allow efficient emulation on real world computers. In the case of programs without reduction, we use PRAM – multiprocessors with as many CPU as necessary. This model is imple-

mented by mapping each processor to a process, and then by multiplexing several processes on one physical processor. On some architectures, this transformation is directly taken care of by the run-time system under the name *virtual processor looping*. In Section 4, we argue that the appropriate model in the presence of reductions is a computer with an unlimited number of processors with unlimited fan-in.

Section 5 describes how the algorithms of [Fea92a, Fea92b] can be adapted to programs with reductions and shows an example of reduction scheduling. A side-effect of reduction detection is that the scheduling process is simplified and improved. This is discussed with some examples in Section 6.

2 The Recurrence Detector

The subject of this paper is the computation of a schedule for programs with explicit reductions. We focus on the programs we obtain by using our recurrences detector presented in [RF93]. Since some readers may be unfamiliar with this work, this section describe which recurrences and reductions are detected by our prototype. For more details on reductions detection, please refer to [RF93] (a state of art is included). Our recurrences detector deals with Systems of Linear Recurrence Equations (SLRE). Indeed, under usual assumptions, a program written in an imperative language (i.e. Fortran) can be translated into a SLRE. A SLRE is a set of equations S. Each equation $e \in S$ is of the form

$$z \in \mathcal{D}_e,\ v_e(z) = \mathrm{Exp}_e(v_{e_1}(I_1(z)), \ldots, v_{e_n}(I_n(z))) \ , \qquad (1)$$

(the set $(v_e)_{e \in S}$ is the family of variables of S). Moreover, we assume that $\mathcal{D}_e$ is a bounded convex. The $(I_k)_{k \in \{1,\ldots,n\}}$ are linear subscripts functions and the functions Exp_e are conditional functions such that

$$\mathrm{Exp}_e(z) = \begin{cases} \mathrm{Exp}_e^1(z) & \text{if } z \in \mathcal{D}_e^1 \\ \vdots & \\ \mathrm{Exp}_e^m(z) & \text{if } z \in \mathcal{D}_e^m \end{cases}$$

(The Exp_e^j are classical mathematical expressions augmented with operators to denote recurrences and the $\mathcal{D}_e^j$ are bounded disjoint convexes). We say that Exp_e is an m clauses expression.

Now, let us present the closed forms we use to denote recurrences. Note that our aim was to detect only the recurrences or the reductions that can be computed efficiently on parallel machines. Hence, only the recurrences whose Detailed Dependences Graph (DDG) is a chain are handled.

The sum of the elements of a matrix can be implemented by:

```
s=0
DO i=1,n
  DO j=1,m
    s=s+a(i,j)
  END DO
END DO
```

The DDG of this program is a chain from the operation corresponding to `(i,j)=(1,1)` to the operation corresponding to `(i,j)=(n,m)`. This reduction can be efficiently computed on parallel machines, for example by using a binary tree like computation. Now, let us consider a SOR computation:

```
DO t=1,max
  DO i=2,n-1
    DO j=2,n-1
      field(t,i,j)=0.25*
        (field(t-1,i-1,j)+field(t-1,i+1,j)+
         field(t-1,i,j-1)+field(t-1,i,j+1))
    END DO
  END DO
END DO
```

The DDG of this program is a tree whose vertices have 4 predecessors. It is useless to detect this recurrence since there is no easy way to implement them efficiently on a parallel machine. Moreover these recurrences are gracefully parallelized using the scheduling technique.

There are two different operators for building closed forms. The first one is a reduction operator (Reduc), it denotes recurrences that can be computed efficiently on a parallel machine (see [RF93]). The second operator (Recur) denotes recurrences that must be computed sequentially. Each value can be computed only when the previous ones are known. In fact, due to the limitations of the recurrence detector some reductions (which can be expressed with the Reduc operator) can be missed and be classified as mere recurrences (so expressed with the Recur operator). Note that the frontier between sequential recurrences and reductions is not a rock solid delimitation since new techniques for computing efficiently some class of recurrences on a parallel machine may be discovered. The Recur operator has two syntaxes, one to express one-dimensional recurrences and one to express multi-dimensional recurrences. To denote an one-dimensional recurrence of order o we use the form

$$\mathrm{Recur}(o, \mathcal{D}, P, f, (g_s)_{s \in \{1, \ \ldots, o\}}) \ .$$

This expression defines a multi-dimensional array of values v computed by the following equation [1]:

$$\forall z \in D,$$
$$v(z) = \begin{cases} \text{if} & P^o(z) \in \mathcal{D} \\ & f(z, v(P^1(z)), \ldots, v(P^o(z))) \\ \text{else if} & P^{o-1}(z) \in \mathcal{D} \\ & f(z, v(P^1(z)), \ldots, v(P^{o-1}(z)), g_o(z)) \\ \vdots & \\ \text{else if} & P^1(z) \in \mathcal{D} \\ & f(z, v(P^1(z)), g_2(z), \ldots, g_o(z)) \\ \text{else} & f(z, g_1(z), \ldots, g_o(z)) \ . \end{cases}$$

Except for multidimensional recurrences we will consider only the case where P is a translation:

$$\forall z \in D,\ P(z) = z + K$$

with K a constant vector. The Recur operator is more complex than reduction operators that can be found in languages such as ALPHA or CRYSTAL because our operator can express general *recurrences* and not only *reductions*. Our Recur operator can express recurrences with order $o > 1$.

For example, a classical computation of the Fibonacci sequence

$$\forall j \in \mathbb{N},\quad v(j) \quad = \quad \begin{cases} 1 \quad \text{if} \quad 0 \leq j \leq 1 \\ v(j-1) + v(j-2) \\ \quad \text{if} \quad 2 \leq j \leq n \ , \end{cases}$$

[1] The notation P^k is used to denote the function $\underbrace{P \circ \cdots \circ P}_{k}$

can be rewritten as

$$\forall j \in \mathbb{N},$$

$$v(j) = \begin{cases} 1 \quad \text{if} \quad 0 \leq j \leq 1 \\ \text{Recur}(\quad 2, \{2 \leq i \leq n\}, \lambda i.i-1, \\ \qquad \lambda ixy.x+y, (\lambda i.1, \lambda \imath.1))(j) \\ \quad \text{if} \quad 2 \leq j \leq n \ . \end{cases}$$

Moreover our operator can deal with recurrences relative to specific dimensions of an array.

For example, the Recur operator can express a set of recurrences relative to the second dimension of a three-dimensional space. A Fortran piece of code to compute such a set of recurrences may be:

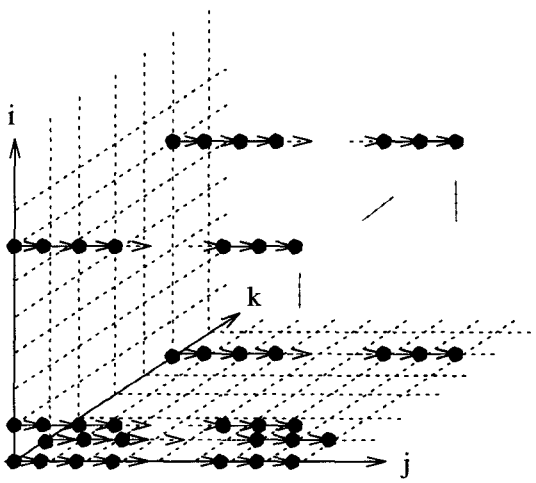
```
v(0:n,0:m,0:1)=a(0:n,0:m,0:1)
DO i=0,n
  DO j=1,m
    DO k=0,1
      v(i,j,k)=v(i,j-1,k)+v(i,j,k)
    END DO
  END DO
END DO
```

An equivalent SLRE is [2]:

$$\forall i,j,k \in \mathbb{N}_n \times \mathbb{N}_m \times \mathbb{N}_l,$$

$$v(i,j,k) = \begin{cases} a(i,0,k) \quad \text{if} \quad j=0 \\ v(i,j-1,k)+a(i,j,k) \\ \qquad \text{if} \quad j \geq 1 \end{cases}$$

The structure of the iteration space of these recurrences is depicted on the picture below (data are transmitted along arrows).



Using the Recur operator this equation can be rewritten as follows:

$$\begin{array}{l} \text{Recur}(\quad 1, \{(0 \leq i_1 \leq n), (0 \leq i_2 \leq m), (0 \leq i_3 \leq l)\}, \\ \qquad \lambda i_1 i_2 i_3.(i_1, i_2, i_3) + (0,-1,0), \\ \qquad \lambda i_1 i_2 i_3 x.x + a(i_1, i_2, i_3), (\lambda i_1 i_2 i_3.0)) \end{array}$$

Even recurrences relative to a combination of dimensions can be expressed by our Recur operator. Indeed the following equation:

$$\forall \imath, j, k \in \mathbb{N}_n \times \mathbb{N}_m \times \mathbb{N}_l,$$

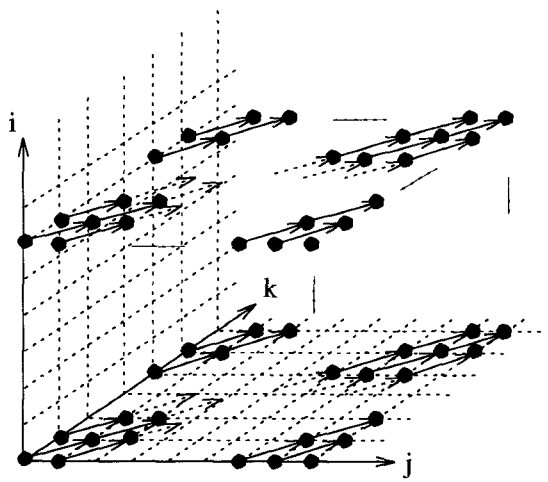
$$v(i,\jmath,k) = \begin{cases} a(i,j,k) \quad \text{if} \quad j=0 \vee k=0 \\ v(i,j-1,k-1)+a(i,\jmath,k) \\ \qquad \text{if} \quad \jmath \geq 1 \wedge k \geq 1 \end{cases}$$

[2] The notation $\mathbb{N}_n$ is used to represent the set $\{0, \ldots, n\}$. In the sequel of this article the notation $\mathbb{N}_n^*$ is also used, this represents the same set as $\mathbb{N}_n$ but excluding 0

has a Recur expression equivalent:

$$\begin{array}{l} \text{Recur}(\quad 1, \{(0 \leq i_1 \leq n), (0 \leq i_2 \leq m), (0 \leq i_3 \leq l)\}, \\ \qquad \lambda i_1 i_2 i_3.(i_1, i_2, i_3) + (0,-1,-1), \\ \qquad \lambda i_1 i_2 i_3 x.x + a(i_1, i_2, i_3), (\lambda i_1 i_2 i_3.0)) \end{array}$$

As shown in the following figure the recurrences are computed along diagonal lines (i.e. following the direction $\vec{j}+\vec{k}$). The initials values of these recurrences are in the planes $(O, \vec{i}, \vec{j})$ and $(O, \vec{i}, \vec{k})$.



Some recurrences are relative to two or more dimensions (e.g. a computation of the sum of the elements of a matrix). These recurrences can be expressed with embedded Recur operators but to make code generation easier we want to denote these recurrences with only one Recur operator. Remark that multi-dimensional recurrences can be expressed with the one-dimensional Recur operator if we allow conditional expressions for the P function. Consider the Recur operator for 1-order recurrences:

$$\text{Recur}(1, \mathcal{D}, P, f, (g)) \ .$$

It computes the following array v:

$$\forall z \in \mathcal{D}, \ v(z) = \begin{cases} \text{if} \quad P(z) \in \mathcal{D} \quad f(z, v(P(z))) \\ \text{else} \quad f(z, g(z)) \ . \end{cases}$$

If we provide a function P which enumerates the iteration space following a lexicographic order the previous equation computes a multi-dimensional recurrence. In order to build this function we use the families of functions $\max_L$ and $\min_L$. Both are functions from a convex $\mathcal{D}$ to the same convex $\mathcal{D}$. If we denote by lexmax the lexicographic maximum, a function $\max_L$ is such that :

$$\max_{l_1,\ldots,l_m}(z) = z - \sum_{\imath=1}^{m} \lambda_\imath.l_\imath \quad \text{where}$$

$$\lambda = \text{lexmax}(\mu \mid \mu \in \mathbb{N}^m, \ z - \sum_{i=1}^{m} \mu_i.l_i \in \mathcal{D}) \quad ,$$

The $\min_L$ functions are defined in a similar way :

$$\min_{l_1,\ldots,l_m}(z) = z + \sum_{\imath=1}^{m} \lambda_\imath.l_\imath \quad \text{where}$$

$$\lambda = \text{lexmax}(\mu \mid \mu \in \mathbb{N}^m, \ z + \sum_{i=1}^{m} \mu_i.l_i \in \mathcal{D}) \quad ,$$

It is then easy to build a $\mathcal{P}$ function to lexicographically iterate a convex domain $\mathcal{D}$ following first the direction l_m then the direction l_{m-1} and so on.

$$\forall z \in \mathcal{D},\ P(z) = \begin{cases} \text{if} & z = \min_{l_1,\ldots,l_m}(z) \\ & z \\ \text{else if} & z = \min_{l_2,\ldots,l_m}(z) \\ & \max_{l_2,\ldots,l_m}(z + l_2) \\ \vdots & \\ \text{else if} & z = \min_{l_m}(z) \\ & \max_{l_m}(z + l_{m-1}) \\ \text{else} & z + l_m \ . \end{cases}$$

This leads to a new Recur operator which integrates this conditional $\mathcal{P}$ function.

$$\text{Recur}_+(\mathcal{D}, (l_i)_{i \in \mathbb{N}_m^*}, f, g) \ . \tag{2}$$

The operator Recur_+ computes the following multi-dimensional array v,

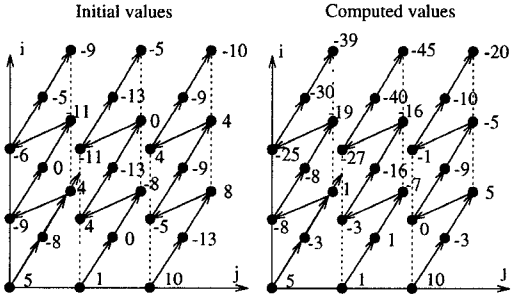
$$\forall z \in D,$$
$$v(z) = \begin{cases} \text{if} & z = \min_{l_1,\ \ldots,l_m}(z) \\ & g(z) \\ \text{else if} & z = \min_{l_2,\ \ldots,l_m}(z) \\ & f(z, v(\max_{l_2,\ldots,l_m}(z + l_2))) \\ \vdots & \\ \text{else if} & z = \min_{l_m}(z) \\ & f(z, v(\max_{l_m}(z + l_{m-1}))) \\ \text{else} & f(z, v(z + l_m)) \ . \end{cases}$$

Note that the operator Recur_+ can express mono-dimensional recurrences of 1-order, it suffices to specify only one direction l_1. That is not a surprise since the operator Recur_+ is a generalization of the operator Recur for order 1 recurrences. We are now able to express recurrences iterated following more than one dimension in a kind of lexicographic order.

For example consider the set of recurrences computed by the following equation:

$$\forall i,j,k \in \mathbb{N}_n \times \mathbb{N}_m \times \mathbb{N}_l,$$
$$v(i,j,k) = \begin{cases} a(0,j,0) & \text{if} \quad i = 0 \wedge k = 0 \\ v(i-1,j,l) + a(i,j,0) & \\ \qquad\qquad \text{if} \quad i \geq 1 \wedge k = 0 & \\ v(i,j,k-1) + a(i,j,k) & \\ \qquad\qquad \text{if} \quad i \geq 1 \wedge k \geq 1 \ . & \end{cases}$$

These recurrences follow the third and then the first dimension of a three-dimensional space. The following figures give an example of the computation with $n = m = l = 3$.



An equivalent expression is given by the following Recur_+ expression:

$$\text{Recur}_+ \quad (\{(0 \leq i_1 \leq n), (0 \leq i_2 \leq m), (0 \leq i_3 \leq l)\}, \{(-1,0,0), (0,0,-1)\}, \lambda i_1 i_2 i_3 x.x + a(i_1,i_2,i_3), \lambda i_1 i_2 i_3.a(0,i_2,0) \ .$$

Note that it is possible to express with the Reduc_+ operator multi-dimensional recurrences iterated following any given directions, one is not bounded to canonical vectors.

The Reduc operator can be defined with the help of the Recur operator. An expression built with an one-dimensional Reduc operator has the following syntax:

$$\text{Reduc}(\mathcal{D}, P, b, d, g) \ , \tag{3}$$

where b is an associative binary function. Such an expression is strictly equivalent to:

$$\text{Recur}(1, \mathcal{D}, P, \lambda i_1 \ldots i_n x.b(x, d(i_1, \ldots, i_n)), (g)) \ . \tag{4}$$

A multi-dimensional Reduc operator may be defined similarly:

$$\text{Reduc}_+(\mathcal{D}, (l_1, \ldots, l_m), b, d, g) \ , \tag{5}$$

is equivalent to

$$\text{Recur}_+(\mathcal{D}, (l_1, \ldots, l_m), \lambda i_1 \ldots i_n x.b(x, d(i_1, \ldots, i_n)), g) \ . \tag{6}$$

3 Basic Scheduling Techniques

Consider a recurrence equation like (1). Provided that every function in Exp_e is strict, we need the values of the $v_{e_k}(I_k(z))$ with $k \in \mathbb{N}_n^*$ in order to compute $v(z)$, and these values must have either been computed beforehand, or be given as initial values of the recurrence. Let $\theta(e,z)$ be the time at which $v_e(z)$ is computed. If we suppose that:

1. the computation of $v_e(z)$ according to (1) can be done in one time unit,
2. our parallel computer has as many processors as necessary,

then $\theta(e,.)$ is a solution of the following *causality condition*:

$$\forall k \in \mathbb{N}_n^*,\ \theta(e,z) \geq \theta(e_k, I_k(z)) + 1 \tag{7}$$

θ is usually called a schedule. This notion was originally introduced in relation with the automatic design of systolic arrays [Qui88]. It has been recently noticed that schedules may also be used to construct parallel programs [Pug91, RWF90]; that point will be further discussed in Section 4.

All techniques for computing schedules may be summarized in the following way:

- Select *a priori* the form of θ. Hardly never, θ is a linear combination of *basis functions* with arbitrary coefficients.
- Substitute the chosen form into (7) and translate these inequalities into constraints on the unknown coefficients.
- Select a set of coefficients which meet the constraints and optimize some objective function, like total latency or delay.

The power of the method depends on the set of basis functions. As far as we know, the most general choice is the one made in [Fea92a, Fea92b]. The basis functions are derived from the inequalities which bound the domains D_e. Constraints on the coefficients are constructed with the help of

Farkas lemma and are linear. Lastly, since most interesting objective functions are linear, the selection of the actual schedule is done by solving a linear program.

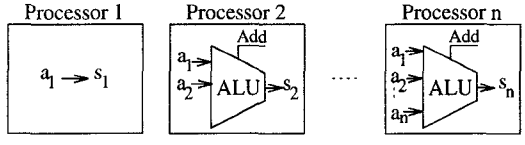
A schedule θ being given, the set of calculation to be done at time t is called a front. If we identify a set of calculation to the set of values it generates, we may write:

$$\mathcal{F}(t) = \{v_e(z) \mid \theta(e,z) = t\} \ .$$

4 Parallel Machine Model

The classical model of parallel machine on which schedulers are based is a shared memory machine with an infinite number of processors. With this model there is no need of constraints on the number of operations in the fronts and the computation of a schedule becomes easier. A similar simplification must be found in order to compute schedules in the presence of reductions. This simplification can be achieved by supposing that the target parallel machine has processors with an infinite fan-in ALU. In this perspective the most common reductions can be performed in constant time.

> On such a machine the classical scan expression,
>
> $$\text{scan}(+, [a_1, \ldots, a_n])$$
>
> which computes the partial sums of the elements of an array can be performed in constant time.



Obviously, a schedule for a machine with an infinite number of processors and with an infinite fan-in ALU per processor must be adapted to work on a realistic machine. Let us consider first the problem of a finite number of processors. The skeleton of the parallel program associated to the schedule:

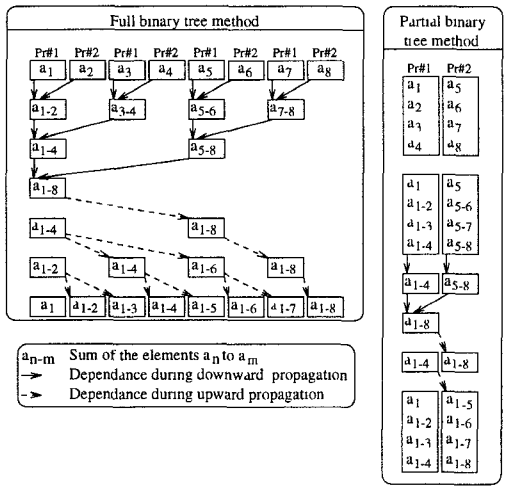
```
DO t=0,L
 Execute all operations of the front number t
END DO
```

is not suitable since a front can include more operations than available processors. The solution is to use a virtualization loop. The parallel program becomes (p is the number of available processors):

```
DO t=0,L
 Let Ops be the set of the operations of the
  front number t
 DO WHILE Ops is not empty
   Execute p operations of Ops
   Delete these operations from Ops
 END DO
END DO
```

Now, let us take into account the fact that a physical machine does not have infinite fan-in ALUs. Reductions and scans are usually computed using a binary tree structure. Hence, if the operation is done on a vector of size n then the computation requires n processors. To perform this computation on a physical machine, a solution is to use again a virtualization loop. But this loop is more complex than the one needed to deal with the finite number of processors. Moreover this second virtualization loop may be inefficient on some type of machines because of the cost of the access to the shared memory. So it seems wise to apply a variant of the binary tree method. This method consists in dividing the data vector into sub-vectors (one vector per available processor), then each processor performs the scan on its private data. A binary tree is used to propagate the partial results. And lastly each processor updates its results.

> Suppose that we want to compute the following scan on a 2 processors machine
>
> $$\text{scan} \quad (+, [a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8]) = \text{Reduc}(\{(1 \le i \le 8)\}, \lambda i.i - 1, +, \lambda i.a_i, \lambda i.0) \ .$$
>
> The two methods are illustrated on the following picture.



With this method for performing scans, the skeleton of the parallel program is:

```
DO t=0,L
 Let Ops be the set of the atomic operations of
  the front number t
 Let Scn be the set of the scan operations of
  the front number t
 DO WHILE Ops is not empty
   Execute p operations of Ops
   Remove these operations from Ops
 END DO
 DO WHILE Scn is not empty
   Perform a scan of Scn using p sub-vectors
   Remove this scan from Scn
 END DO
END DO
```

If Scn includes scans on vectors with size less than p then some processors will be idle in the scan loop. But scans operate usually on large data vectors. Moreover it is possible to optimize this parallel program by gathering scans on small data in the same iteration of the scans loop. This way of handling scans is somewhat basic, but sufficient to demonstrate that a reasonably efficient parallel program can be generated from our schedule. More sophisticated implementation of scans can be used, for example the Harmonic Schedule presented in [NW91]. Practically, we plan to use the existing scan primitives of the target machines (every parallel machine but a network of workstations has such primitives). In fact, every scan computation algorithm can be used with no important change of the parallel program skeleton.

Note that the two hypothesis on the theoretical machine (infinite number of processors and infinite fan-in ALU) lead to the same order of delay. Indeed a front with n atomic operations on a parallel machine with only p processors can be handled in $\frac{n}{p}$ time units and a front with one scan on a size n data vector can be handled in $\frac{n}{p} + \log_2 p$ time units. Even with a machine with 64K processors (i.e. a Connection Machine 2) we only have a difference of 16 time units.

5 Scheduling Algorithms Modifications

In the output system of the recurrence detector one can find new expressions which come from the resolution of trivial recurrences or reductions (expressed with the Reduc or the Reduc_+ operators). Mere recurrences are very important for purposes such as reducing access to memory or optimizing the use of registers, but for this preliminary work we suppose that expressions built with the Recur or the Recur_+ operators cannot appear in the system to schedule. To get rid of the recurrences one can come back to the original expressions of the recurrences. Such a transformation is implemented in our prototype, but will not be described here.

5.1 Trivial Recurrences

We define the trivial recurrences as recurrences which have a numerically exact solution. By numerically exact solution we mean an expression which gives values at least as accurate as the values computed by the initial sequential algorithm.

For example, the following recurrence $\mathcal{R}$,

```
a(1)=alpha
a(2)=beta
DO i=3,n
  a(i)=a(i-2)
END DO
```

has a numerical exact solution which is

```
DO i=1,n
  IF (MOD(i,2).EQ.1) THEN
    a(i)=alpha
  ELSE
    a(i)=beta
  END IF
END DO
```

But the Fibonacci sequence, which can be implemented as follows,

```
a(1)=1
a(2)=1
DO i=3,n
  a(i)=a(i-1)+a(i-2)
END DO
```

has an algebraic solution which includes exponentiations. Hence this recurrence has no numerical exact representation. For example with the IEEE representation of floating point values on 32 bits the computation of the solution

$$\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^i - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^i$$

differs by one unit from the correct value for $i = 32$.

In fact no modification is needed to compute schedules in the presence of trivial recurrences. Indeed the recurrence detector replaces these recurrences by their solutions.

If the input of the recurrence detector is fed with the recurrence $\mathcal{R}$, the output is the following system:

$$\left[\begin{array}{lcl} v1 & = & \texttt{alpha} \\ v2 & = & \texttt{beta} \\ \forall i \in \mathbb{N}_n - \{0,1,2\}, & & \\ v3(i) & = & \text{if } (i \bmod 2) = 1 \text{ then } \texttt{alpha} \text{ else } \texttt{beta} \end{array}\right.$$

The algorithms from [Fea92a, Fea92b] can be directly applied to this example.

5.2 Reductions

To schedule reductions expressed with the Reduc operator some modifications must be applied to the algorithms from [Fea92a, Fea92b]. These modifications concern the generation of causality conditions. Consider a general Reduc expression, with a general predecessor function P (to address both cases of one and multi-dimensional reductions):

$$\forall z \in \mathcal{D},\ v(z) = \text{Reduc}(\mathcal{D}, P, b, d, g)(z)\ . \tag{8}$$

For each point z of D, we denote by $\mathcal{P}(z)$ the set

$$\{z' \in \mathcal{D} \mid \exists \alpha \in \mathbb{N},\ z' = P^\alpha(z)\}\ .$$

Intuitively, this set includes the points of $\mathcal{D}$ associated with data (the $d(z')$) needed to compute the value $v(z)$. We distinguish a special point $\mathcal{G}(z)$ (the initialization point, which is not in $\mathcal{D}$). This point is characterized by

$$\exists \alpha \in \mathbb{N}^*,\ \mathcal{G}(z) = P^\alpha(z) \notin \mathcal{D} \text{ and } P^{\alpha-1}(z) \in \mathcal{D}\ .$$

In this context, the causality conditions for equations like (8) are:

$$\begin{array}{l} \forall z \in \mathcal{D},\ \forall z' \in \mathcal{P}(z),\ \theta(v,z) \geq \theta(d,z') + 1\ , \\ \forall z \in \mathcal{D},\ \theta(v,z) \geq \theta(g, \mathcal{G}(z)) + 1\ . \end{array} \tag{9}$$

Note that $\theta(d,.)$ and $\theta(g,.)$ represent the times at which the data and the initialization values are ready. Actually, the causality conditions will have one inequality for each reference to another equation in the expressions d and g. The previous causality conditions express the fact that, on a computer with an unlimited number of processors and with an unlimited fan-in, one gets the result from a reduction one clock step after the time all necessary data are available.

Causality conditions like (9) are not of the form (7). However, the solution methods from [Fea92a, Fea92b] may be adapted provided that the set

$$\{(z,z') \mid z \in \mathcal{D} \text{ and } z' \in \mathcal{P}(z)\}\ , \tag{10}$$

is convex. Let

$$\forall k \in \mathbb{N}^*_{n_v},\ \Phi_{v,k}(z) \geq 0\ ,$$

be the set of inequalities which defines the domain $\mathcal{D}_v$ of v. By Farkas Lemma, since $\theta(v,.)$ is non negative in $\mathcal{D}_v$, we may write

$$\theta(v,z) = \mu_0 + \sum_{k=1}^{n_v} \mu_k \Phi_{v,k}(z) \text{ with } \forall k \in \mathbb{N}^*_{n_v},\ \mu_k \geq 0\ .$$

Similarly, let

$$\forall k \in \mathbb{N}^*_{n_\mathcal{P}},\ \Psi_{\mathcal{P},k}(z,z') \geq 0\ ,$$

be the inequalities which define the set described by (10). Again by Farkas Lemma:

$$\theta(v,z) - \theta(d,z') - 1 \equiv \lambda_0 + \sum_{k=1}^{n_{\mathcal{P}}} \lambda_k \Psi_{\mathcal{P},k}(z,z') \ .$$

This is an identity in which coefficients of like variables may be equated. The resulting set of equations in the λ's and the μ's characterizes the set of valid schedules. A particular schedule may then be selected in this set by optimizing some criteria, like latency or delay. The reader is referred to [Fea92a] for details.

The difficulty lies in the computation of the sets $\mathcal{P}(z)$ and $\{\mathcal{G}(z)\}$. Note that all sets that include these two sets give valid schedules, albeit possibly with a greater latency or delay. The simplest approximation is to take $\mathcal{D}$ for $\mathcal{P}(z)$ and $\mathcal{P}(\mathcal{D}) - \mathcal{D}$ [3] for $\{\mathcal{G}(z)\}$. This gives the following causality conditions:

$$\forall z \in \mathcal{D},\ \forall z' \in \mathcal{D},\ \theta(v,z) \geq \theta(d,i) + 1 \ ,$$
$$\forall z \in \mathcal{D},\ \forall z' \in \mathcal{P}(\mathcal{D}) - \mathcal{D},\ \theta(v,z) \geq \theta(g,z') + 1 \ .$$

These conditions are easy to obtain but they can be improved. Two cases occur, one-dimensional and multi-dimensional reductions. The problem is easier with the one-dimensional case since the predecessor function P is a translation, $P(z) = z + K$:

$$\mathcal{P}(z) = \{z' \in \mathcal{D} \mid \exists \alpha \in \mathbb{N},\ z' = z + \alpha K\} \ .$$

This set is equivalent to

$$p_1(\{(z',\alpha) \in \mathcal{D} \times \mathbb{N} \mid z' = z + \alpha K\}) \ ,$$

where p_1 represents the projection along the first dimension. To obtain a convex the variable α must be eliminated from equations $z' = z + \alpha K$. That is always possible since the vector K is non null. If there are no coefficients in K equal to 1, which occurs very seldom, the resulting convex will be rational. But the approximation is acceptable since the rational convex will include $\mathcal{P}(z)$. There is no such an easy way to determine the vertex $\mathcal{G}(z)$. The more practicable method is to use a Parametric Integer Programming method (see [Fea88]). We use the software PIP to resolve the following problem:

$$\alpha_0 = \mathrm{Min}\{\alpha \mid \alpha \in \mathbb{N},\ z + \alpha K \in \mathcal{D}\}$$

In this context $\mathcal{G}(z) = z + (\alpha_0 - 1)K$.

Let us consider, now, a general multi-dimensional reduction

$$\mathrm{Reduc}_+(\mathcal{D}, (l_1, \ldots, l_m), b, d, g) \ .$$

For this reduction, the set $\mathcal{P}(z)$ can be expressed as follows:

$$\{z' \in \mathcal{D} \mid z' =_{l_1,\ldots,l_m} z \wedge z' \ll_{l_1,\ldots,l_m} z\} \ ,$$

where the operator $=_{l_1,\ldots,l_m}$ represents the fact that all coordinates of the two vectors must be equal but for dimensions l_1 to l_m. The operator $\ll_{l_1,\ldots,l_m}$ represents a lexicographic order with respect only to the dimensions l_1 to l_m. This set can be considered as the union of the following sets:

$$\mathcal{P}_{l_1}(z) = \{z' \in \mathcal{D} \mid z' =_{l_1,\ldots,l_m} z \wedge z'_{l_1} < z_{l_1}\}$$
$$\mathcal{P}_{l_2}(z) = \{z' \in \mathcal{D} \mid z' =_{l_1,\ldots,l_m} z \wedge z'_{l_1} = z_{l_1} \wedge z'_{l_2} < z_{l_2}\}$$
$$\vdots$$
$$\mathcal{P}_{l_m}(z) = \{z' \in \mathcal{D} \mid z' =_{l_1,\ldots,l_m} z \wedge z'_{l_1} = z_{l_1} \wedge \ldots \wedge z'_{l_{m-1}} = z_{l_{m-1}} \wedge z'_{l_m} \leq z_{l_m}\} \ .$$

[3] $\mathcal{P}(\mathcal{D}) - \mathcal{D}$ is not a convex domain in general. It must be decomposed into convexes to be usable by Farkas Lemma.

Each of these sets is convex. Hence the causality condition concerning $\mathcal{P}(z)$ can be described as the conjunction of m inequalities on convex domains. To obtain the vertex $\mathcal{G}(z)$, PIP is used to find the lexicographic minimum (in the sense of $\ll_{l_1,\ldots,l_m}$) of the set $\mathcal{P}(z)$. The vertex $\mathcal{G}(z)$ is this lexicographic minimum except that one must subtract 1 to the coordinate l_1.

Most of the time, a linear subscript is applied to the array of values computed by the reduction:

$$\forall z \in \mathcal{D}',\ v(z) = \mathrm{Reduc}(\mathcal{D}, P, b, d, g)(\phi(z)) \ . \qquad (11)$$

In this case the causality conditions become:

$$\forall z \in \mathcal{D}',\ \forall z' \in \mathcal{P}(\phi(z)),\ \theta(v,z) \geq \theta(d,z') + 1 \ ,$$
$$\forall z \in \mathcal{D}',\ \theta(v,z) \geq \theta(g, \mathcal{G}(\phi(z))) + 1 \ .$$

Moreover an expression of an equation can include embedded reductions. In this case an auxiliary schedule must be associated to each internal reduction. Causality conditions are then computed for each reduction and the whole system is solved. The auxiliary schedules do not appear in the final result. This method is equivalent to isolate internal reductions into new equations and then generating the causality conditions.

As an example of scheduling computation for embedded reductions consider the following program:

```
DO i=1,n
 DO j=1,m
  a(i,j)=a(i,j-1)+a(i,j)
 END DO
END DO
DO i=1,n
 DO j=1,m
  a(i,j)=a(i-1,j)+a(i,j)
 END DO
END DO
```

Our recurrence detector gives the following result:

$$\begin{array}{l}
\forall (i,j) \in \mathbb{N}_n \times \mathbb{N}_m, \\
\quad v1(i,j) = \\
\qquad \mathrm{Reduc}(\\
\qquad\quad \{(0 \leq i_1 \leq n), (0 \leq i_2 \leq m)\}, \\
\qquad\quad \lambda(i_1,i_2).(i_1,i_2) + (-1,0), \\
\qquad\quad +, \\
\qquad\quad \lambda i_1 i_2.\mathrm{Reduc}(\\
\qquad\qquad\quad \{(0 \leq j_1 \leq n), (0 \leq j_2 \leq m)\}, \\
\qquad\qquad\quad \lambda(j_1,j_2).(j_1,j_2) + (0,-1), \\
\qquad\qquad\quad +, \\
\qquad\qquad\quad \lambda j_1 j_2.v4(j_1,j_2), \\
\qquad\qquad\quad \lambda j_1 j_2.v3(i_1,i_2+1))(i_1,i_2), \\
\qquad\quad \lambda i_1 i_2.v2(j_1+1,j_2))(i,j) \ ,
\end{array}$$

where $v1$ represents the values in $a(i,j)$ at the end of the computation and $v2$, $v3$ and $v4$ are initial values in a before the calculation starts. Let us consider first the outer reduction, denoting by $\theta(e,.)$ the schedule of the inner reduction. The set $\mathcal{P}(z)$ is obtained from the intersection of

$$\mathcal{D} = \{(0 \leq i_1 \leq n), (0 \leq i_2 \leq m)\} \ ,$$

and the projection of the following convex along the α axis.

$$\{i', j', \alpha \mid i' = i - \alpha \wedge j' = j\} \ .$$

This intersection is

$$\{i', j' \mid 0 \leq i' \leq i \wedge j' = j\} \ .$$

Therefore, the causality conditions for the external Reduc expression are:

$$\begin{array}{l}\forall(i,j) \in \mathbb{N}_n \times \mathbb{N}_m,\ \forall i' \in \mathbb{N}_t, \\ \theta(v1,(i,j)) \geq \theta(e,(i',j)) + 1\ , \\ \forall(i,j) \in \mathbb{N}_n \times \mathbb{N}_m, \\ \theta(v1,(i,j)) \geq \theta(v2,(0,j)) + 1\ .\end{array}$$

We compute now causality conditions for the internal reduction (that will give us the θ_e schedule). The conditions are:

$$\begin{array}{c}\forall(i,j) \in \mathbb{N}_n \times \mathbb{N}_m, \forall j' \in \mathbb{N}_j, \\ \theta(e,(i,j)) \geq \theta(v4,(i,j')) + 1\ , \\ \forall(i,j) \in \mathbb{N}_n \times \mathbb{N}_m, \\ \theta(e,(i,j)) \geq \theta(v3,(i,0)) + 1\ .\end{array}$$

To apply the Farkas Lemma on these causality conditions, we must know the schedules of equations $v1$, $v2$ and $v3$ (computed in a previous stage). Suppose that initializations are done at time 0 and that the values of a are computed following the schedule $\theta(v4,(i,j)) = m - j + i$. The new system to resolve becomes:

$$\begin{array}{l}\forall(i,j) \in \mathbb{N}_n \times \mathbb{N}_m, \forall j' \in \mathbb{N}_j, \\ \theta(e,(i,j)) \geq m - j' + i + 1\ , \\ \forall(i,j) \in \mathbb{N}_n \times \mathbb{N}_m,\ \forall i' \in \mathbb{N}_i, \\ \theta(v1,(i,j)) \geq \theta(e,(i',j)) + 1\ .\end{array}$$

Since the schedules are non negative in the domain of their equation, the application of the Farkas Lemma gives:

$$\begin{array}{c}\forall(i,j) \in \mathbb{N}_n \times \mathbb{N}_m, \\ \theta(e,(i,j)) = \mu_0 + \mu_1 i + \mu_2(n-i) + \mu_3 j + \mu_4(m-j)\ ,\end{array}$$

and

$$\begin{array}{c}\forall(i,j) \in \mathbb{N}_n \times \mathbb{N}_m, \\ \theta(v1,(i,j)) = \mu'_0 + \mu'_1 i + \mu'_2(n-i) + \mu'_3 j + \mu'_4(m-j)\ ,\end{array}$$

(the scalars $\mu_0,\ldots,\mu_4$ and $\mu'_0,\ldots,\mu'_4$ are non negative). With another application of the Farkas Lemma to the first causality condition we obtain:

$$\begin{array}{ll}\mu_0 + \mu_1 i & +\mu_2(n-i) + \mu_3 j \\ & +\mu_4(m-j) - (m - j' + i) - 1 \\ \equiv & \\ \lambda_0 + \lambda_1 i & +\lambda_2(n-i) + \lambda_3 j + \lambda_4(m-j) \\ & +\lambda_5 j' + \lambda_6(j - j')\ .\end{array}$$

Therefore, one has to resolve the following linear system, with some optimization criteria to find an optimal schedule:

$$\begin{array}{lcl}\mu_1 - \mu_2 & = & \lambda_1 - \lambda_2 + 1 \\ \mu_3 - \mu_4 & = & \lambda_3 - \lambda_4 + \lambda_6 \\ 0 & = & \lambda_5 - \lambda_6 - 1 \\ \mu_2 & = & \lambda_2 \\ \mu_4 & = & \lambda_4 + 1 \\ \mu_0 & = & \lambda_0 + 1\ .\end{array}$$

By hand, it is a good idea to try the lowest values for the μ's. This often leads to good schedules. The previous system may be simplified:

$$\begin{array}{lcl}\mu_1 & = & \lambda_1 + 1 \\ \mu_3 & = & \lambda_3 + \lambda_6 + 1 \\ 0 & = & \lambda_5 - \lambda_6 - 1 \\ \mu_2 & = & \lambda_2 \\ \mu_4 & = & \lambda_4 + 1 \\ \mu_0 & = & \lambda_0 + 1\ .\end{array}$$

The constraints on the μ's are:

$$\mu_0 \geq 1,\ \mu_1 \geq 1,\ \mu_2 \geq 0,\ \mu_3 \geq 1,\ \mu_4 \geq 1\ .$$

Taking the lowest possible values gives for the inner reduction the schedule $\theta(e,(i,j)) = i + m + 1$. This is the expected result. A similar analysis concerning the first causality condition would lead to the schedule $\theta(v1,(i,j)) = i + m + 2$ for the whole equation.

6 Conclusion

Reduction loops in scientific computation programs are a source of parallelism which has not been efficiently used in present day compilers. We have shown that this can be done by a three steps approach:

- Normalize the program and identify recurrences;
- Detect those recurrences which are in fact reductions, i.e. which are built from an associative operator;
- Schedule the resulting system, taking into account the fact that the arguments of a reduction may be computed in any order.

The resulting parallelization method, which has been implemented as a new phase in the PAF parallelizing compiler, has one main advantage, that of finding parallelism in apparently sequential kernels, and two subsidiary ones, accelerating the scheduling computation and giving better schedules.

Since the recurrence detector is a reduction algorithm it tends to transform input program into a single equation. In fact in most cases this cannot be achieved because of the constraints presented in [RF93]. But the number of equations is always reduced with respect to the initial SLRE built from the source program. Hence, since the number of vertices in the data-flow graph of the resulting SLRE is smaller than the number of vertices in the data-flow graph of the source program, the computation of the schedule for the SLRE is usually faster. And sometimes, detecting the recurrences and computing an SLRE schedule is considerably faster than computing the schedule for the initial program. This can be observed when the data-flow of the source program includes large strong components and the recurrence detector reduces these strong components to smaller ones. This is due to the fact that the scheduler solves linear systems whose sizes are proportional to the size of the strong components. Therefore the scheduling run-time increases rapidly with the size of the strong components.

We have made some tests with programs like the following.

```
DO i=1,n
 a1(i)=f1(c1(i-1))        (v1)
 a2(i)=f2(c1(i-1))        (v2)
 a3(i)=f3(c1(i-1))        (v3)
 a4(i)=f4(c1(i-1))        (v4)
 b1(i)=g1(a1(i),a2(i))    (v5)
 b2(i)=g2(a3(i),a4(i))    (v6)
 c1(i)=h1(b1(i)+b2(i))    (v7)
END DO
```

The recurrence detector reduces this type of program to only one equation. Therefore the scheduler has to build a system corresponding to a strong component with only one vertex, not a system corresponding to a strong component with 7 vertices. We made tests with 4, 8 and 16 levels of auxiliary variables. Below are given the run-times to schedule directly the programs and the run-times to first detect recurrences and then schedule the resulting system of equations.

Level	Direct scheduling	Recurrences detection and Equations scheduling
4	15s	10s+1s
8	100s	20s+1s
16	820s	45s+1s

Moreover, since we associate a linear function to each clause (and not to each equation, that is to each instruction) our schedule can be better than the schedule computed for the original program (even if the program does not include reductions). Better means that the latency of the schedule is smaller. In fact computing schedules for clauses leads to compute better linear piecewise schedules.

Consider the following loop

```
x(0)=0                (v1)
DO i=1,2*n
  x(i)=x(2*n-i+1)   (v2)
END DO
```

A direct scheduling gives the following schedules for **v1** and **v2**:

$$\theta(v1) = 0,\ \forall \imath \in \mathbb{N}^*_{2n},\ \theta(v2, i) = i - 1$$

The recurrence detector transforms the initial program into a two clauses equation.

$$\forall i \in \mathbb{N}^*_{2n},\ v2(i) = \begin{cases} x(2n - \imath + 1) & \text{if } \imath \leq n \\ v2(2n - \imath + 1) & \text{if } i \geq n + 1 \end{cases}$$

The scheduling of the *clauses* gives trivial schedules.

$$\forall i,\ 1 \leq i \leq n,\ \theta(\mathrm{c1}, i) = 0,\ \forall i,\ n{+}1 \leq i \leq 2n,\ \theta(\mathrm{c2}, \imath) = 1$$

It now remains to generate efficient code from the schedules thus constructed. This will be left for future work. Another point to be studied is enlarging the knowledge base on reductions and trivial recurrences. A larger base may produce more *apparent* parallelism, but this advantage may be offset by more complicated elementary operations – for instance replacing scalar operations by a matrix product – or loss of precision, as in the case of the Fibonacci example. More work is needed to see if we have struck the right balance here.

References

[Cal91] D. Callahan. Recognizing and parallelizing bounded recurrences. In U. Banerjee et al. (Eds.), editor, *Proc. of the Fourth International Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 266–282. Springer-Verlag, August 1991. LNCS 589.

[Fea88] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part i, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] Paul Feautrier. Somme efficient solution to the affine scheduling problem, part ii, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, December 1992.

[Ive62] Kenneth A. Iverson. *A Programming Language*. Jonh Wiley & Sons, New York, 1962.

[Mau89] Christophe Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. PhD thesis, Université de Rennes I, December 1989.

[MCL88] Y.-I. Choo M. Chen and J. Li. Crystal: From functional description to efficient parallel code. In G. Fox, editor, *Proc. of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 417–433. ACM, New York, USA, 1988.

[NW91] A. Nicolau and H. Wang. Optimal schedules for parallel prefix computation with bounded resources. In *Third ACM SIGPLAN Symposium on Principles and and Pratice of Parallel Programming PPOPP*, April 1991.

[Pug91] William Pugh. Uniform techniques for loop optimization. *ACM Conf. on Supercomputing*, pages 341–352, January 1991.

[Qui88] Patrice Quinton. Mapping recurrences on parallel architectures. In *3rd Int. Conf. on Supercomputing*, Boston, May 1988.

[RF93] X. Redon and P. Feautrier. Detection of reductions in sequentials programs with loops. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *Procs of the 5th International Parallel Architectures and Languages Europe*, pages 132–145, June 1993.

[RWF90] Mourad Raji-Werth and Paul Feautrier. Systematic construction of programs for distributed memory systems. In Paul Feautrier and François Irigoin, editors, *Procs of the Int. Workshop on Compiler for Parallel Computers, Paris*, December 1990.