# COMPILING FOR MASSIVELY PARALLEL ARCHITECTURES: A PERSPECTIVE

Paul FEAUTRIER
*Laboratoire PRiSM*
*Université de Versailles Saint-Quentin*
*45 Avenue des Etats-Unis,*
*78035 VERSAILLES CEDEX FRANCE*
*Paul.Feautrier@prism.uvsq.fr*

ABSTRACT: The problem of automatically generating programs for massively parallel computers is a very complicated one, mainly because there are many architectures, each of them seeming to pose its own particular compilation problem. The purpose of this paper is to propose a framework in which to discuss the compilation process, and to show that the features which affect it are few and generate a small number of combinations. The paper is oriented toward fine-grained parallelization of static control programs, with emphasis on dataflow analysis, scheduling and placement. When going from there to more general programs and to coarser parallelism, one encounters new problems, some of which are discussed in the conclusion.

KEYWORDS Massively Parallel Compilers, Automatic Parallelization.

```
@ARTICLE{Feau:95,
         AUTHOR = {Paul Feautrier},
         TITLE ={Compiling for Massively Parallel Architectures: a
  Perspective},
         JOURNAL = {Microprogramming and Microprocessors},
         YEAR = 1995,
         NOTE = {to appear}
        }
```

# 1 A FRAMEWORK FOR DISCUSSING MASSIVELY PARALLEL COMPILATION

The problem of automatically generating programs for massively parallel computers is a very complicated one, mainly because there are many architectures, each of them seeming to pose its own particular compilation problem. The purpose of this paper is to propose a framework in which to discuss the compilation process, and to show that the features which affect it are few and generate a small number of combinations.

We will first introduce some notations for discussing parallel programs. We will then restate in our framework several classical techniques: data expansion, scheduling, partitioning, tiling and loop rewriting. It is then possible to explore the spectrum of parallel architectures, and to show that each of them may be programmed by one of the above techniques, or by a combination of them. In the conclusion, we will point to several short range and long range unsolved problems.

## 1.1 Static Control Programs

An *operation* of a program is one execution of an instruction. While the number of instructions is roughly proportional to the size of the program text, the number of operations is proportional to the running time of the program and may vary according to the size of the data. What is to be taken as an instruction depends on the purpose of the analysis. In the case of source-to-source parallelization, which is our main concern here, we will identify instructions with simple statements in the source high level language. Other choices are possible, e.g. in the case of Instruction Level Parallelization.

At present, parallelization techniques apply only to static control programs, i.e. programs for which one may describe at compile time the set of operations which are going to be executed in a program run. Static control programs are built from assignment statements and `DO` loops. In such a program, the only mechanism which generates operations from instructions is `DO` loop iteration. As a consequence, an operation may be named by a tuple $\langle S, \vec{x} \rangle$ where $S$ is the name – or label – of the parent instruction, and where the components of $\vec{x}$ – the iteration vector – are the values of the surrounding loop counters, from outside inward. The dimension of $\vec{x}$ (i.e. the number of loops which surround $S$), will be denoted $N_S$.

The only data structures are arrays of arbitrary dimension. For technical reasons, loop bounds and array subscripts are restricted to affine forms in the loop counters and integral structure parameters, which are assumed to be known at program loading. From each of the surrounding `DO` statements, we may extract an upper and a lower bound for the corresponding counter. Under the above hypotheses, these bounds may be collected as a system of $2N_S$ affine constraints:

$$D_S \vec{x} + \vec{d}_S \geq 0.$$

The integer solutions of these inequalities define the iteration domain $\mathcal{D}_S$ of $S$. Such a set is known as a $\mathsf{Z}$-polyhedron.

Generally, the result of a program depends on the order in which its operations are

executed. The fact that operation $u$ is executed before operation $v$ is written $u \prec v$. If several programs are under discussion, their *execution orders* will be distinguished by subscripts. A sequential program is associated to a total execution order. A parallel program is associated to a partial execution order. For the purpose of understanding what is the result of a parallel program, one may assume that, for each run, target computer selects in some way a total extension of this partial order and executes it sequentially. Since in general a partial order has many total extensions, it follows that a parallel program has many possible executions, with potentially many different results. The central question of automatic parallelization is: when do all these executions give the same result?

Two operations $u$ and $v$ are *independent* if their order of execution can be reversed without changing the global effect on the program store. Let $\mathcal{R}(u)$ and $\mathcal{M}(u)$ be the sets of memory locations which are read and modified by operation $u$. A sufficient condition for independence of $u$ and $v$ is [Ber66]:

$$\mathcal{M}(u) \cap \mathcal{M}(v) = \emptyset, \tag{1}$$

$$\mathcal{R}(u) \cap \mathcal{M}(v) = \emptyset, \tag{2}$$

$$\mathcal{M}(u) \cap \mathcal{R}(v) = \emptyset. \tag{3}$$

If these conditions are not satisfied, the operations $u$ and $v$ are said to be *dependent*, written $u \,\delta\, v$.

Suppose that two programs have the same set $E$ of operations. One of them is sequential, with total execution order $\prec$. The other one has execution order $\prec_{//}$, presumably parallel. One may show that a sufficient condition for the equivalence of these two programs is:

$$\forall u, v \in E : u \,\delta\, v \wedge u \prec v \Rightarrow u \prec_{//} v, \tag{4}$$

in words, that if two operations are dependent, then they are executed in the same order in the sequential and the parallel version.

Another equivalent formulation is that the execution order of any correct parallelization must be a transitive extension of the relation $\prec \cap \delta$ , the Detailed Dependence Graph (DDG) of the source program.

As a consequence, the parallel compilation process may be divided in two steps: firstly, compute the DDG of the source program, then select any extension of the DDG which can be executed efficiently on the target architecture.

With the exception of programs without loops – the *basic blocs* – this proposal cannot be carried out litteraly, since the size of the DDG is enormous and may vary from run to run. The concern of most parallelization methods is to construct a summary of the DDG, the objective being to keep just enough information for the construction of a parallel program.

## 2 BASIC ANALYSIS TECHNIQUES

Recent research on automatic parallelization has concentrated on two topics: improving the dependence calculation, and finding well-defined algorithms for the construction of extensions of the DDG.

### 2.1 Array Dataflow Analysis

Each edge in the dependence relation may be seen as a constraint on the final parallel program. There is an obvious interest in removing as many edges as possible. Some edges are related to memory reuse (the so-called output- and anti-dependences) and can be removed by modifying the data structure of the program. The remaining edges represent dataflow from a definition to a use of the same memory cell. However, a definition may be *killed* before being used by another operation.

The aim of array dataflow analysis is to characterize in a compact way the set of proper flow dependences of a program. Let us consider an operation $u$ in which a memory cell $\mathtt{c}$ is used. Let us write $\mathcal{W}(\mathtt{c})$ for the set of operations which write into $\mathtt{c}$. The *source* of $\mathtt{c}$ at $u$ is the latest write into $\mathtt{c}$ which precedes $u$:

$$\mathrm{source}(\mathtt{c}, u) = \max_{\prec}\{v \mid v \prec u, v \in \mathcal{W}(\mathtt{c})\}.$$

When the linearity hypotheses of section 1.1 are taken into account, this problem translates into the calculation of the lexical maximum of a union of $\mathsf{Z}$-polyhedra. To express the result, we need the concept of quasi-affine form : a formula which is built from the variables by the operations of addition, integer multiplication and integer division. One may show [Fea88b] that the source above can be expressed as a piecewise combination of quasi-affine forms in the coordinates of the iteration vector $\vec{x}$ of $u$. In many cases, integer division does not occur, and each piece of the solution may be written:

$$\mathrm{source}(\mathtt{c}, \langle R, \vec{x}\rangle) = \langle S, L_{RS}\vec{x} + \vec{\ell}_{RS}\rangle, \tag{5}$$

where $L_{RS}$ is a matrix and $\vec{\ell}_{RS}$ is a vector of suitable dimensions.

It may happen that some memory cell is not defined in the program fragment under study. In that case, we use the special operation $\perp$ which may be interpreted as the initial program loading. To simplify the exposition, we will suppose here that all initializations have been made explicit in the program text, and hence that $\perp$ never occurs in sources. There are practical algorithms for computing the source function, with the help, e.g., of linear programming [Fea88a, Fea91] or by simplifying Pressburger formulas [PW93]. It happens frequently that the source can be found directly without resorting to linear programming techniques. See [MAL93] or [HT94] for a study of such cases.

Consider the following matrix multiplication code:

```
      do i = 1,n
        do j = 1,n
1         a(i,j) = ...
2         b(i,j) = ...
        end do
      end do
      do i = 1,n
        do j = 1,n
3         c(i,j) = 0.0
          do k = 1,n
4           c(i,j) = c(i,j) + a(i,k)*b(k,j)
```

```
      end do
    end do
  end do
```

Let us investigate the source of `c(i,j)` in operation $\langle 4, i, j, k\rangle$. The candidates are statement 3 and statement 4 itself. Let us consider an operation $\langle 4, i', j', k'\rangle$. A candidate must write into the proper memory location, which implies $i' = i, j' = j$, and be executed earlier than $\langle 4, i, j, k\rangle$, which, together with the preceding equalities, implies $k' < k$. The latest possible source is thus $\langle 4, i, j, k-1\rangle$. This operation exists only if $k \geq 2$. If $k = 1$, a similar reasoning indicates that the source is $\langle 3, i, j\rangle$. These results may be put together as a conditional:

$$\text{source}(\texttt{c(i,j)}, \langle 4, i, j, k\rangle) = \textbf{if } k \geq 2 \textbf{ then } \langle 4, i, j, k-1\rangle \textbf{ else } \langle 3, i, j\rangle, \tag{6}$$

The other sources are simpler:

$$\text{source}(\texttt{a(i,k)}, \langle 4, i, j, k\rangle) = \langle 1, i, k\rangle, \tag{7}$$

$$\text{source}(\texttt{b(k,j)}, \langle 4, i, j, k\rangle) = \langle 2, k, j\rangle. \tag{8}$$

When the source function is known, a new version of the equivalence condition (4) can be stated. Let $\mathcal{R}(u)$ be the set of memory cells which are used by operation $u$. One must have:

$$\forall u, \forall \texttt{c} \in \mathcal{R}(u) : \text{source}(\texttt{c}, u) \prec_{//} u, \tag{9}$$

in words, that the source of a value in any operation is executed before that operation in the parallel version of the program. The graph whose vertices are the operations, with an edge from $v$ to $u$ iff $v$ is the source of a value which is used by $u$, is the Dataflow Graph (DFG) of the program.

However, using any execution order which satisfies (9) for constructing a parallel program will give an incorrect result, because output- and anti-dependences have not been taken into account. One can get rid of these dependences by data expansion. We will suppose here, for simplicity, that each operation returns only one result. Let us associate to each operation $u$ one distinct memory cell $M[u]$. Let us write the statement associated to $u$ as:

$$\texttt{a} := f(\ldots, \texttt{c}, \ldots). \tag{10}$$

Consider the program in which operation $u$ execute the following statement:

$$M[u] := f(\ldots, M[\text{source}(\texttt{c}, u)], \ldots). \tag{11}$$

Since each operation $u$ is executed only once, and since the result location $M[u]$ is in one-to-one correspondence with $u$, this program has the single assignment property. When the program starts, all memory cells are undefined. The cell $M[u]$ gets a value when $u$ is executed, and this value does not change until the end of the program. One may prove that this single assignment program, executed according to any order $\prec_{//}$ which satisfies (9) is equivalent to the original sequential program, in the following sense:

**Theorem 1** *The value which is computed by operation $u$ is the same in the original program (10) and in the parallel version (11).*

It is clear that the sequential execution order $\prec$ is well founded. Hence, we may use induction on this order to prove equivalence. Let $u$ be an operation, and suppose that equivalence has been proved for all $v \prec u$. Let $\mathtt{c}$ be a memory cell in $\mathcal{R}(u)$. Its value has been generated by $\text{source}(\mathtt{c}, u) \prec u$. By (9), $\text{source}(\mathtt{c}, u)$ has been executed before $u$ in the parallel program, and the resulting value, which is written into $M[\text{source}(\mathtt{c}, u)]$, is, by the induction hypothesis, equal to the value generated by $\text{source}(\mathtt{c}, u)$ in the sequential program. This value is never overwritten by the single assignment property. As a consequence, the function $f$ has the same arguments in (10) and in (11), and hence, gives the same results, QED.

Single assignment programs have the property that their space complexity and time complexity are the same. This is in contrast to the situation which prevails for most scientific computing algorithms, where the space requirement is an order of magnitude less than the time requirement. On the other hand, implementing an algorithm as a single assignment program allows one to extract all the parallelism of which it is susceptible. In many cases, this degree of parallelism may be attainable with less than total memory expansion, and beside, this degree of parallelism may not be necessary, e.g. because the target computer cannot make use of it. It would be nice in these cases to be able to adjust the amount of memory expansion, but this is still an open research subject.

### 2.2 Scheduling

The problem is now to specify an execution order for the parallel program, i.e. an extension of the DDG or of the DFG. The specification of an order on a large set is very complex. Our aim here is to find simple representations, even if we have to sacrifice some parallelism in order to achieve simplicity. The use of a *schedule*, i.e. of a function which maps the set of operations to logical time (i.e. to any linearly ordered set) is such a simple representation.

To any function $\theta$ mapping the set of operation to an ordered set, we may associate the partial order:

$$u \prec_\theta v \equiv \theta(u) < \theta(v).$$

Usually, the range of $\theta$ is taken to be the integers. $\theta$ is a valid schedule if the corresponding order satisfies (4) or (9), i.e:

$$u \,\delta\, v \wedge u \prec v \Rightarrow \theta(u) < \theta(v), \tag{12}$$

or

$$\forall u, \forall \mathtt{a} \in \mathcal{R}(u) : \theta(\text{source}(\mathtt{a}, u)) < \theta(u). \tag{13}$$

The *latency* of a schedule is the maximum value of the schedule over the set of operations. It may be interpreted as the running time of the parallel program on a PRAM. It is clear that the source relation is included in the DDG. Hence, (12) is a tighter constraint than (13). As a consequence, the valid schedules according to (13) have a latency no larger than that of schedules which satisfy (12). The price to pay is that if a schedule for (13) is used, the data space has to be expanded in order to restore correctness.

In the case of our running example, the following functions:

$$\theta(1,i,j) = 0, \quad \theta(2,i,j) = 0, \quad \theta(3,i,j) = 0, \quad \theta(4,i,j,k) = k.$$

are valid schedules. Consider for instance the constraint associated to memory cell `c(i,j)` in statement 4. We have to check that:

$$(\textbf{if } k \geq 2 \textbf{ then } \theta(4,i,j,k-1) \textbf{ else } \theta(3,i,j)) < \theta(4,i,j,k).$$

The test splits into two subproblems. If $k \geq 2$, we have $k - 1 < k$. If $k < 1$, then, from the loop lower bound we deduce that $k = 1$, and the condition to be verified is that $0 < 1$. The reader may care to test the other conditions on $\theta$.

To solve the scheduling inequalities, one starts by postulating a simple prototype for $\theta$:

$$\theta(S, \vec{x}) = \vec{h}_S . \vec{x} + k_S,$$

where $\vec{h}_S$ is known as the *timing vector* for $S$. These prototypes for $\theta$ are substituted into (12) or (13). From the result one may deduce linear inequalities on the unknowns components of $\vec{h}_S$ and $k_S$.

There are several methods for deducing these inequalities. One may give a set of cleverly chosen values to $\vec{x}$ (the vertices of the iteration domains [Qui87]) or apply Farkas lemma [Fea92a].

Most of the time, these linear constraints have many solutions. One usually selects a "best" one according to some figure of merit. One possibility is to optimize the latency of the program [DR95]. Alternately, one may search for earliest start time or *leftmost* schedules [Fea92a]. In both cases, the solution is obtained by solving linear programming problems. The authors of [KP94] have attempted to optimize more complicated objective functions by a search process.

It may happen that the set of affine constraints which is deduced from the dependence conditions (12) or (13) has no solution. A possibility in that case is to construct multi-dimensional schedules by the algorithm in [Fea92b] which has been proved optimal in the case of uniform dependences in [DV94].

To any function $\theta$, one may associate the system of disjoint sets:

$$\mathcal{F}(t) = \{u \mid \theta(u) = t\}. \tag{14}$$

$\mathcal{F}(t)$ is the *front* at time $t$. If $\theta$ satisfies (12), it is clear that all operations inside a front are independent, i.e. can be executed in parallel. If the schedule satisfies (13), then the source and the sink of any value do not belong to the same front: there is no data exchange inside a front. We may say that the set of operations has been partitioned into *anti-chains* – sets of non comparable operations – which are executed sequentially. This is the `SEQ of PAR` style of programming of [Bou93]. Two operations in different fronts are executed in sequence, but are not necessarily in dependence: some parallelism has been lost in the interest of a simple parallel program representation.

### 2.3 Distribution

Let $\mathcal{Q}$ be any partition of the set of operations of a given program. One may associate to $\mathcal{Q}$ an execution order in the following way. Operations which belongs to the same part of $\mathcal{Q}$ are executed according to the sequential execution order. Operations which belong to different parts are ordered in the sequential order if and only if they are in dependence. It is quite clear that the resulting order satisfies (4). Intuitively, each part corresponds to a process. Ordering between operations in different parts necessitates a synchronization operation. This is the `PAR of SEQ` style of programming of [Bou93].

In the same fashion, one may introduce an ordering between two operations in different parts only if one is the source and the other a sink for a given value. In that case, the ordering is obtained by transmitting a message from the source to the sink; the message carries the shared value.

**Theorem 2** *The program obtained by partitioning and synchronization is correct, provided that the workspace of the original program is replicated in all processors.*

> We have to prove that the value which is computed by an operation $u$ is the same in the sequential and in the distributed version. Here again, the proof is by induction on the sequential execution order. Let us suppose that the theorem is true for all operation $v \prec u$. Let $\mathtt{c}$ be a memory cell in $\mathcal{R}(u)$; in the distributed version, this cell is replicated in all processors. There are two cases:
>
> - source$(\mathtt{c}, u)$ belongs to the same part as $u$. By construction, the source operation has been executed *before* $u$ in the distributed version, and has left the correct value in $\mathtt{c}$. Beside, this value has not been obliterated in the distributed version, because this would contradict the definition of the source function.
> - source$(\mathtt{c}, u)$ belongs to a different part. We have here to make some assumptions about the communication mechanism. Suppose for simplicity that as soon as the source operation terminates, it broadcasts a message holding its name and its result. When $u$ initiates, it waits for a message holding the name of the source operation and gets the value contained therein. Since source$(\mathtt{c}, u) \prec u$, the value obtained in this way is the correct one by the induction hypothesis.
>
> All in all, the arguments of $u$ are the same in the sequential and in the distributed case, hence its result is the same, QED.
>
> The communication mechanism which is postulated here relies on a kind of tuple space *à la* Linda [CG89]. There are other possibilities, see section 4.3.

Here, *any* partition gives a correct parallel program. However, efficiency considerations dictate that the set of residual synchronisation or communications be kept to a minimum, subject to the condition that all processes execute about the same amount of work. The problem has been widely studied [LC91, AL93, Fea94]. A plausible solution is the following. One postulates a *placement function* $\Pi$ from the set of operations to the set of processes (also called virtual processors). The virtual processors are understood to occupy the points of a $g$-dimensional $\mathsf{Z}$-polyhedron (the *template* of HPF, the *geometry* of the CM-2, and so on). $\Pi(u)$ is a function from the set of operations to $\mathsf{Z}^g$ which gives the coordinates of the

virtual processor which executes $u$. For each value which is used by $u$, one may define a communication distance:

$$d(\mathtt{a}, u) = \Pi(\text{source}(\mathtt{a}, u)) \perp \Pi(u) \tag{15}$$

and the equation $d(\mathtt{a}, u) = 0$ expresses the fact that the source for cell $\mathtt{a}$ in operation $u$ is in the same process as $u$. If all such equations can be satisfied, all residual communications will disappear.

Since an arbitrary placement function is useless for program restructuring purposes, one makes the additional assumption that $\Pi$ is affine:

$$\Pi(R, \vec{x}) = \vec{\pi}_R.\vec{x} + q_R,$$

where $\vec{\pi}_R$ is an unknown vector and $q_R$ an unknown alignment constant. Let us recall formula (5):

$$\text{source}(\mathtt{c}, \langle R, \vec{x} \rangle) = \langle S, L_{RS}\vec{x} + \vec{\ell}_{RS} \rangle.$$

If we substitute this formula and the above prototype placement function into (15), then identify, we get a system:

$$\begin{aligned} \vec{\pi}_S L_{RS} &= \vec{\pi}_R, \\ \vec{\pi}_S.\ell_{RS} + q_S &= q_R. \end{aligned}$$

The first line expresses the fact that the communication distance is a constant (a very desirable property), and the second line that this constant is null. Let us write $\vec{\pi}$ for a vector in which all unknowns $\vec{\pi}_R$ for all $R$ are collected. Similarly, let $\vec{q}$ be the vector of all alignment constants. The above system may be summarized as:

$$\begin{aligned} C\vec{\pi} &= 0, \\ D\begin{pmatrix} \vec{\pi} \\ \vec{q} \end{pmatrix} &= 0, \end{aligned}$$

where $C$ is known as the *communication matrix* of the program. The meaning of the first line is that $\vec{\pi}$ must belong to the null space of $C$ [BKK+94]. If, as is likely for real world programs, $C$ is of full row rank, $\vec{\pi} = 0$ is the only solution, and the calculation collapses on processor 0. To obtain an interesting solution, one needs $g$ linearly independent placement functions, where $g$ is the dimension of the processor grid. This means that one has to select a submatrix of $C$ with a $g$-dimensional null space. The excluded rows corresponds to residual communications. Heuristics may be used to select the excluded communications among those with the lightest load [Fea94]. When $\vec{\pi}$ has been selected, one tries to satisfy as many alignment conditions as possible, thus obtaining *local communications*.

> In some cases, it is possible to solve the placement equations without any hypotheses on the form of the placement functions. To the source functions (6–8) are associated the following placement equations:
>
> $$\Pi(4, i, j, k) = \Pi(4, i, j, k-1), \tag{16}$$
> $$\Pi(4, i, j, 1) = \Pi(3, i, j), \tag{17}$$
> $$\Pi(4, i, j, k) = \Pi(1, i, k), \tag{18}$$
> $$\Pi(4, i, j, k) = \Pi(2, k, j). \tag{19}$$
>
> From (16) we deduce that $\Pi(4, i, j, k)$ does not depend on $k$. Similarly, (18–19) imply that this function does not depend on either $j$ or $i$, and hence is a constant. It

is thus impossible to build a distributed program for our example without residual communications.

Suppose now we ignore (19). We may now take:

$$\Pi(4,i,j,k) = \Pi(1,i,j) = \Pi(3,i,j) = i.$$

Equations (16–18) are satisfied. We may choose $\Pi(2,i,j)$ arbitrarily. Let us take $\Pi(2,i,j) = j$. There are then two solutions. The first one is to program a communication from processor $j$ to processor $i$ in which `b(k,j)` is sent. The second one is to duplicate `b` on all processors. There is no residual communication in this case.

In writing (15), it has been supposed that the value which is generated by $u$ is held in the memory of processor $\Pi(u)$. This is the well known "owner computes rule". Relaxing this rule may give more efficient placements, see [DR94].

A linear placement function for a program whose iteration domain has characteristic dimension $n$ has a range of cardinality $O(n)$, and hence, generates $O(n)$ processes. This may be too much for some architectures. In that case, one *folds* the placement function by assigning several processes to one physical processor. Alternatively, $O(n)$ may not be enough for some architectures like the CM2 or Maspar. In that case one uses two or more linearly independent placement functions. Such a $g$-dimensional placement function generates $O(n^g)$ processes, $g$ being limited only by the dimension of the iteration space.

One can compute a placement function without any reference to a schedule. However, there are two reasons not to do that. The first one is that knowing the schedule allows one to choose the dimensionality of the placement. If the iteration space has dimension $d$, and if the schedule is one dimensional, then each front is included in a subspace of dimension $d \perp 1$, and there is no need to use a processor grid of higher dimension. More generally, for each statement the maximum dimension of the grid which allows full utilisation of the processors is $d \perp s$ where $s$ is the dimension of the schedule. Secondly, the schedule and placement function seen as a space-time transform has to be one-to-one, meaning that each process executes at most one operation at any given time:

$$\Pi(u) = \Pi(v) \wedge \theta(u) = \theta(v) \Rightarrow u = v. \tag{20}$$

### 2.4 Supernode Partitioning

In this technique [IT88], one starts again with a partition $\mathcal{S}$ of the operation set. The elements of $\mathcal{S}$ are called *supernodes* or *tiles*. This partition is subjected to the requirement that the quotient of the dependence graph by $\mathcal{S}$ is acyclic:

$$\forall \sigma, \tau \in \mathcal{S},\ \nexists u, v \in \sigma,\ x, y \in \tau : u \prec x, y \prec v, u\,\delta\,x, v\,\delta\,y.$$

In the parallel version of the program, operations which belong to the same supernode are executed sequentially according to $\prec$, while supernodes themselves are executed according to the quotient order. Supernode partitioning is an important technique for improving the performance of a parallel program, by adapting the "grain of parallelism" of the program to the grain of the target computer. Most often, supernodes are defined as identical *tiles* which have to cover the set of operations of the program. As a first approximation, the

computing time of a tile is of the order of its *volume*, while the necessary communications or synchronization are of the order of its *surface*. Increasing the size of tiles improves the computation to communication ratio, at the price of reducing the amount of parallelism. The extreme is the case of only one tile, which generates no communication and no parallelism. The problem of writing the actual parallel program after tiling is simply displaced from the original dependence graph to the quotient graph, and the methods above still apply.

Some loop nests contain *fully permutable loops* [WL91] i.e. loops which can be arbitrarily reordered without change in the results of the nest. The problem of tiling is much simpler in that case.

## 3 LOOP REWRITING

The aim of the methods that have been discussed in the preceding section is to expose the parallelism in the source program. There is still the problem of rewriting the program in such a way that this parallelism may be easily exploited by, e.g., the native compiler or run-time system.

> To the system of fronts (14), we may associate the following program:
>
> ```
> do t = 0, L
>     doall F(t)
> end do
> ```
>
> However, in this program, the `doall` is purely a notational convention. If this code is to be submitted to a real compiler, we have to find a parametric representation of $\mathcal{F}(t)$ in term of $t$ and of a set of new variables, and to construct a set of loops for the enumeration of $\mathcal{F}(t)$. We know in advance that these loops will be parallel.

In full generality, the problem may be expressed in the following terms [KP92]. We are given a one-to-one mapping $\mathcal{T}$ from the set of operation to $\mathbb{N}^d$:

$$\mathcal{T}(u) = \mathcal{T}(v) \Rightarrow u = v. \tag{21}$$

The execution order $\prec_{\mathcal{T}}$ which is associated to $\mathcal{T}$ is the lexicographic order on $\mathcal{T}(u)$:

$$u \prec_{\mathcal{T}} v \equiv \mathcal{T}(u) \ll \mathcal{T}(v).$$

The selection of $\mathcal{T}$ is the important step in the parallelization process. As we will see in the next section, it is strongly influenced by the target architecture. In particular, by a clever choice of $\mathcal{T}$, it will happen that some of the components of $\mathcal{T}(u)$ may be interpreted as processor numbers, the corresponding loops being parallel. Other components may be interpreted as logical time, giving sequential loops. However, the loop rewriting process is completely independent of these considerations.

The $\mathcal{T}$ transformation is not arbitrary. In fact, as we will see later, it is made up by combining in various ways affine schedules and affine placement functions. The consequence

is that the restriction of $\mathcal{T}$ to any statement $S$ is an affine transform from the iteration domain of $S$ to $\mathsf{N}^d$:

$$\mathcal{T}(S, \vec{x}) = T_S \vec{x} + \vec{r}_S, \tag{22}$$

where $T_S$ is a $d \times N_S$ matrix. For the condition (21) to be satisfied, $T_S$ has to be of rank $N_S$.

Loop rewriting in general is a very complicated process. It is helpful to start by solving several simpler subproblems. The basic problem is, a $\mathsf{Z}$-polyhedron being given as a set of inequalities:

$$\mathcal{D} = \{\vec{x} \mid D\vec{x} + \vec{d} \geq 0\},$$

to construct a loop nest which scans $\mathcal{D}$ in lexicographic order. The first solution has been given in [Iri87] (see also [AI91]), using an extension of the Fourier-Motzkin pairwise elimination method.

> Basically, let $x_d$ be the counter of the innermost loop. One rewrites a row of $D$ in which $x_d$ has a positive coefficient as a lower bound on $x_d$. Similarly, a row where the coefficient is negative gives a lower bound. The lower bound of $x_d$ is the maximum of all the lower bounds found in this way. An upper bound is found in a similar fashion. One then eliminates $x_d$, and the process is repeated for all counters from inside outward.

Other solutions use parametric linear programming [CFR94] or the Chernikova algorithm for constructing the vertices of a polyhedron [VWD94].

The next problem is the one in which the loop nest is subjected to a transformation associated to a unimodular matrix $T$. Let $T^{-1}$ be its integral inverse. In that case, the iteration domain after the transformation is still a $\mathsf{Z}$-polyhedron, which is given by the inequalities:

$$DT^{-1}\vec{y} + \vec{d} \geq 0.$$

The new loop bounds are found by one of the above methods.

If $T$ is not unimodular, the image of the iteration domain is no longer a $\mathsf{Z}$-polyhedron. The first step is to compute the left Hermite normal form of $T$ [Min83]:

$$T = HQ,$$

where $Q$ is unimodular. Since $T$ is of rank $N_S$, $H$ is lower triangular with positive elements. Let us introduce an auxiliary integral vector $\vec{z}$ such that:

$$\vec{z} = Q\vec{x},$$

$$\vec{y} = H\vec{z}.$$

The special form of $H$ implies that $\vec{y}$ is a monotone increasing function of $\vec{z}$. Since $Q$ is unimodular, we may find a loop nest wich scans the $\mathsf{Z}$-polyhedron $Q\mathcal{D}_S$ by the above method. This loop nest is then rewritten in term of $\vec{y}$ by applying the matrix $H$. In particular, the diagonal elements of $H$ give the steps of the new loop nest [Dar93, Ris94, Xue94].

The most complicated case is the one in which we have to rewrite several statements with different transformations. Each transformation has the same target space and is supposed to have full rank. However, it is not necessary to suppose that the whole transformation is one to one. If two operations (or more) coming from different statements are scheduled at the same time and place, it is a simple matter to have them executed in an arbitrary order.

We have to scan the union of the various images of iteration domains. However, it is well known that the union of several polyhedra is not necessarily a polyhedron. One possibility is to scan the convex hull of this union, inserting guards to avoid executing non-existant operations. The compiler should be careful to detect trivial guards in order to reduce overhead. The other possibility is to dissect the union of iteration domains into an union of disjoint polyhedra, and to write a loop nest for each subset. The drawback of this method is code duplication. The reader is referred to the original publications for details [Col94, AALL93, KP92].

After rewriting the loop nests, one still has to modify the statements themselves. This can be done in two ways. The simplest solution is to express the old variables $\vec{x}$ in term of the new one $\vec{y}$ by inverting (22), which is always possible by (20). The values of $\vec{x}$ are then substituted into the array subscripts of $S$.

When working with the single assignment form (11), there is a more interesting possibility, called *reindexing*. Let us introduce a new data space, $N$, which is indexed by the transformed coordinates $v = \mathcal{T}(u)$:

$$\begin{aligned} N[v] &= M[u], \\ N[\mathcal{T}(u)] &= M[u], \\ N[v] &= M[\mathcal{T}^{-1}(v)]. \end{aligned}$$

(11) is reindexed into:

$$N[v] = f(\ldots, N[\mathcal{T}(\text{source}(\mathsf{c}, \mathcal{T}^{-1}(v))], \ldots) \tag{23}$$

As we shall see later, when $\mathcal{T}$ is constructed from a schedule and/or a placement, this form has specially interesting properties.

# 4 ADAPTING THE COMPILER TO THE ARCHITECTURE

## 4.1 Classifying Architectures and Languages

It is a truism that each programming language defines – sometime explicitly, most of the time implicitly – an underlying virtual architecture. In many cases, the user of a massively parallel computer only sees the virtual architecture provided by his favorite programming language. This leads to the distinction between the programming model and the execution model [Bou93]. In this discussion, we will mostly stay at the level of the programming model. For instance, any computer which runs Fortran 90 will be deemed a vector processor.

Obviously, when constructing programs for massively parallel computers, one has to take the target architecture into account. My contention is that only broad characteristics of the target computer are important for the compiling process. Detailed parameters, like e.g. message latencies or cache size, are to be taken into account only when fine tuning the

resulting program, as for instance when one has to decide the size of supernodes.

The main characteristics of a parallel architecture are the following:

- Is there a central clock which synchronizes all processors?
- Is there a global address space which can be accessed in a uniform manner by all processors?

These two parameters are largely independent, and thus gives rise to four architectural classes.

### 4.2 Global Memory Synchronous Architectures

Under this category fall static control superscalar and VLIW processors, and also a few designs like Opsila (a global memory SIMD machine [ABD90]). Parallelism is obtained by executing a large number of operations simultaneously at each clock cycle. One may argue that pipeline processors belong to this class, if one stays at the level of the vector instructions, and one ignores the detailed programming of the pipelines.

For a synchronous computer, each operation has a well defined date, which is obtained simply by counting clock cycles from the beginning of the program. This gives a natural schedule. Conversely, to a given schedule, one may associate the following abstract synchronous program:

```
do t = 0, L
   doall F(t)
end do
```

where $L$ is the *latency* of the schedule. The body of the loop may be understood as a very large instruction, each processor taking charge of one of its elementary operations. This program cannot in general be executed directly. Firstly, in a synchronous computer, all operations in a front are to be instances of the same instruction. Secondly, the size of the front is limited by the number of identical processors. One has to split the front into subfronts $\mathcal{F}_S$ according to the statement $S$ which is executed. One also has to adjust the schedule in such a way that no front has more operations than the available number of processors. This can be integrated into the scheduling process, or done *a posteriori* by variants of the well-known strip mining technique, or left to the run-time system.

The code generation process for this case may be explained simply in term of loop rewriting. Let us suppose first that the schedule for statement $S$, $\theta_S$ is one dimensional and is defined by a primitive timing vector[1] $\vec{h}_S$. One extends $\vec{h}_S$ to a unimodular matrix by constructing its Hermite normal form [Dar93]. A more complicated process is needed when the timing vector is not primitive or when the schedule is multidimensional, see [Col94].

If the schedule has been computed from the dataflow graph, one has to do some form of data expansion to obtain a correct program. Single assignment conversion is usually too

[1] A vector is primitive iff its coordinates are mutually prime integers.

much expansion. The problem of finding the minimum expansion which still gives a correct parallel program is a very important one, see [MAL93, Cha93]. A partial solution may be obtained by reindexing. If the first row of $\mathcal{T}$ is given by a schedule, the shape of (23) is:

$$N[t, \ldots] = f(\ldots, N[t \perp d, \ldots], \ldots),$$

where $t$ is logical time and $d$ is a positive delay by (13). If the delays for the various statements have an upper bound $D$, then the first subscript of $N$ may be folded modulo $D+1$ without introducing unwanted dependence. This process usually reduces data expansion to more manageable proportions.

### 4.3 Distributed Memory Asynchronous Architectures

This is a class of computers with a very large population, from workstation networks to hypercube based architectures. Each processor works in asynchrony and has its own independent memory. A message exchange is necessary if one processor needs a value which has been computed elsewhere, and, since message passing is always much slower than computation, such exchanges must be kept to a minimum.

These computers are best programmed from a partition as in Section 2.3. To a first approximation, the data space of the original program can be replicated in each memory, thus insuring the needed data expansion. Each processor runs a copy of the source program, each instruction being guarded to insure that it is only executed if it has been assigned to that processor. The overall effect is SPMD programming.

Let us suppose that distribution is specified by a placement function $\Pi$, and let $q$ be the current processor number. Operation $u$ is replaced by the following code [ZBG88]:

$\forall \mathtt{a} \in \mathcal{R}(u)$ : `if` $\Pi(u) \neq q \wedge \Pi(\text{source}(\mathtt{a}, u)) = q$ `then Send(a) to` $\Pi(u)$
`if` $\Pi(u) = q \wedge \Pi(\text{source}(\mathtt{a}, u)) \neq q$
`then Receive(a) from` $\Pi(\text{source}(\mathtt{a}, u))$
`if` $\Pi(u) = q$ `then` $\mathtt{c} = f(\mathcal{R}(u))$

One may prove that there is a one-to-one correspondence between Sends and Receives, that values are sent in the order in which they are received, and that Send-Receive pairs implement the needed synchronization between operations in different processors.

The efficiency of the above scheme can be improved in several ways. Firstly, as many guards as possible should be pushed up into the surrounding loop bounds. This can be done only for simple forms of the placement function. Secondly, proper choice of the placement function should minimize the number of residual communications. Sends and Receives should be grouped in order to have longer messages, perhaps by supernode partitioning.

Nevertheless, any partition function leads to a correct if perhaps inefficient object program. This is the reason why it is feasible to have the distribution specified by the programmer, as in the HPF language.

### 4.4 SIMD Architectures

SIMD architectures, as for instance the CM-2 or Maspar computers, or systolic arrays, have synchronous processors *and* a distributed memory. Hence, for generating a parallel program, one needs both a schedule and a placement function which together have to satisfy constraint (20). Once these functions have been found, they are put together as the first rows of the space-time transformation $\mathcal{T}$. If the template is of small dimension (for instance, because the target computer is a linear array) it may be necessary to complete the transformation by adding extra loops, which are to be executed sequentially by the processing elements. Most of the time, the resulting transformation will not be unimodular, and the more complicated algorithms of section 3 have to be used.

Data expansion poses the same problem here as in the global memory case. In addition, one has to construct communications statements for residual communications. Here again, reindexing is the key to the solution. The components of $\mathcal{T}(\text{source}(\mathtt{c}, \mathcal{T}^{-1}(v))$ which corresponds to processor numbers give the *routing transformation* for operation $v$. A specially efficient case is that in which the routing depends only on the processor numbers. If the routing is one to one, it has to be implemented as a bulk communication. If the routing is defined by a translation, this communication is a neighbor to neighbor shift. If the routing is not one to one, it may be implemented as a *broadcast* [RWF91].

The question of satisfying (20) is very difficult. Experience shows that actual solutions of the placement and scheduling constraints usually have the right property. Whether this is a coincidence or a feature of our algorithms is unknown at present.

Some authors have proposed that scheduling and placement be coupled in some way. One possibility is to select a placement first, then to take the data transfer time into account when computing a schedule. One rewrites (13) in the form:

$$\forall u, \forall \mathtt{a} \in \mathcal{R}(u) : \theta(\text{source}(\mathtt{a}, u)) + 1 + \tau(u, \text{source}(\mathtt{a}, u)) \leq \theta(u). \tag{24}$$

$\tau(u, \text{source}(\mathtt{a}, u))$ is the transfer time from processor $\Pi(\text{source}(\mathtt{a}, u))$ to processor $\Pi(u)$. It is zero if the source and sink operations are executed by the same processor, and depends in a complicated way on the network topology and possibly on simultaneous communications (through contention phenomena) if not.

> The only situation in which an analogous problem has an easy solution is that of systolic arrays. In that case, the communications are "pushed up" among the computations by a transformation known as *uniformization* [QD89]. The transformed algorithm is then scheduled in the usual way. Since the communication network is custom built according to the structure of the transformed algorithm, no contention is ever possible. Attempts to transpose this paradigm to SIMD architectures did not give satisfactory results, perhaps because general purpose communication networks are quite different from systolic array networks.

### 4.5 Asynchronous Shared Memory Architectures

An asynchronous multiprocessor with a global address space is apparently the easiest parallel architecture to program. In fact, there are two ways of tackling the job. Firstly, it is

easy to emulate a synchronous architecture – one needs only a fast `barrier` primitive – and still easier to emulate a distributed memory machine: one has only to partition the address space and to restrict access of each processor to the associated memory segment. This is the policy most operating systems implement for data security reasons. The memory access limitation is raised for the time it takes to execute communication primitives.

However, one should be aware that today global memories are build on top of message passing architectures, either as a Shared Virtual Memory, or with the help of distributed caches. In both cases, the performance of the computer is sensitive to the placement of the data and calculations. The most important parameter is the coherence protocol, which insures that no obsolete data is returned to a read request.

Strong coherence protocols [CF78] work by invalidating extra copies of data before allowing a modification. In that case, the main concern is to avoid coherence induced cache misses. This is done, not by distributing the data, but by distributing the operations, two operations sharing the same datum being preferably located on the same processor.

Weak coherence protocols delimit sections of the code where it is safe to waive the coherence check because there are no concurrent writes to the same memory cell. Observe that a front is such a section, because all operations inside it have the single assignment property. Coherence is restored as a side effect of the `barrier` operation. It is clear that fronts are exactly what is needed to construct weakly coherent sections of the code, and that a schedule is the natural tool for constructing fronts.

## 5 CONCLUSION

This paper has attempted to give guidelines for the central part of massively parallel program synthesis in the case of static control programs. This has to be preceded by an analysis phase and followed by a loop rewriting phase. The construction of the DFG and scheduling are well understood processes. Distribution and placement is a fuzzier subject since there is no real constraint on the placement function, the problem being one of trading off load balancing against communications. Some work is still needed in that direction.

The basic tools for code generation are well understood. There are, however complex interactions between code generation, memory usage optimization and communication implementation, which are still largely unexplored. The problem is complicated by the fact that the choice of an optimal solution is strongly dependent of the structure of the run time system (see e.g. [RWF95] for a discussion of the influence of the so-called *vp-looping* scheme on the performances of the CM-2).

The next step is to go beyond static control programs, exploring `while` loops and irregular data structures. There is still some hope of improving analysis and synthesis techniques in this direction [CBF95]. However, we are nearing the point at which the information content of the program text is nearly exploited to the full. After this stage, the only possibilities are either run-time parallelization methods or the definition of new programming languages, where more information is available for parallelization.

## ACKNOWLEDGMENT

Many thanks to Luc Bougé, who carefully criticized a first version of this paper.

## References

[AALL93] Saman P. Amarasinghe, Jennifer M. Anderson, Monica S. Lam, and Amy W. Lim. An overview of a compiler for scalable parallel machines. In *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, pages 253–272. Springer Verlag, LNCS 768, August 1993.

[ABD90] Michel Auguin, Fernand Boéri, and Jean-Paul Dalban. Synthèse et évaluation du projet OPSILA. *TSI*, 9:79–98, 1990.

[AI91] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proc. third SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 39–50. ACM Press, April 1991.

[AL93] Jennifer M. Anderson and Monica S. Lam. Global optimization for parallelism and locality on scalable parallel machines. *ACM Sigplan Notices*, 28:112–125, June 1993.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on El. Computers*, EC-15, 1966.

[BKK+94] David Bau, Indupras Kodukula, Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Solving alignment using elementary linear algebra. In *Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, pages 46–60. Springer-Verlag, LNCS 892, August 1994.

[Bou93] Luc Bougé. Le modèle de programmation à parallélisme de donnés : une perspective sémantique. *T.S.I.*, 12(5):541–562, 1993.

[CBF95] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy array dataflow analysis. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. ACM, July 1995.

[CF78] Lucien M. Censier and Paul A. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. on Computers*, C-27:1112–1118, December 1978.

[CFR94] Jean-François Collard, Paul Feautrier, and Tanguy Risset. Construction of `do` loops from systems of affine constraints. *Parallel Processing Letters*, to appear, 1994.

[CG89] Nicholas Carriero and David Gelernter. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys*, 21(3), September 1989.

[Cha93] Zbigniew Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle*. PhD thesis, IFSIC, Rennes I, February 1993.

[Col94] Jean-François Collard. Code generation in automatic parallelizers. In Claude Girault, editor, *Proc. Int. Conf. on Application in Parallel and Distributed Computing, IFIP WG 10.3*, pages 185–194. North Holland, April 1994.

[Dar93] A. Darte. *Techniques de parallélisation automatique de nids de boucles.* PhD thesis, ENS Lyon, April 1993.

[DR94] Alain Darte and Yves Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Computing*, 20:679–710, 1994.

[DR95] Alain Darte and Yves Robert. Affine-by-statement scheduling of uniform and affine loop nests over parametric domains. *J. Parallel and Distributed Computing*, 1995. to appear.

[DV94] Alain Darte and Frédéric Vivien. Automatic parallelization based on multidimensional scheduling. Technical Report RR 94-24, LIP, 1994.

[Fea88a] Paul Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing*, pages 429–441, 1988.

[Fea88b] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[Fea91] Paul Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.

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

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

[Fea94] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[HT94] C. Heckler and L. Thiele. Computing linear data dependencies in nested loop programs. *Parallel Processing Letters*, 4(3):193–204, 1994.

[Iri87] François Irigoin. *Partitionnement de boucles imbriquées, une technique d'optimisation pour les programmes scientifiques.* PhD thesis, Université P. et M. Curie, Paris, June 1987.

[IT88] François Irigoin and Rémi Triolet. Supernode partitioning. In *Proc. 15th POPL*, pages 319–328, San Diego, Cal., January 1988.

[KP92] Wayne Kelly and William Pugh. Generating schedules and code within a unified reordering transformation framework. Technical Report TR-92-126, Univ. of Maryland, November 1992.

[KP94] Wayne Kelly and William Pugh. Selecting affine mappings based on performance estimations. *Parallel Processing Letters*, 4(3):205–220, September 1994.

[LC91] Jingke Li and Marina Chen. The data alignment phase in compiling programs for distributed memory machines. *Journal of Parallel and Distributed Computing*, 13:213–221, 1991.

[MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array dataflow analysis and its use in array privatization. In *Proc. of ACM Conf. on Principles of Programming Languages*, pages 2–15, January 1993.

[Min83] Michel Minoux. *Programmation Mathématique, théorie et algorithmes*. Dunod, Paris, 1983.

[PW93] William Pugh and David Wonnacott. An evaluation of exact methods for analysis of value-based array data dependences. In *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, pages 546–566. Springer-Verlag LNCS 768, August 1993.

[QD89] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *The Journal of VLSI Signal Processing*, 1:95–113, 1989.

[Qui87] Patrice Quinton. The systematic design of systolic arrays. In F. Fogelman, Y. Robert, and M. Tschuente, editors, *Automata networks in Computer Science*, pages 229–260. Manchester University Press, December 1987.

[Ris94] Tanguy Risset. *Parallélisation Automatique: du modèle systolique au à la compilation des nids de boucles*. PhD thesis, ENS Lyon, February 1994.

[RWF91] Mourad Raji-Werth and P. Feautrier. On parallel program generation for massively parallel architectures. In M. Durand and F. El Dabaghi, editors, *High Performance Computing II*. North-Holland, October 1991.

[RWF95] Mourad Raji-Werth and Paul Feautrier. On factors limiting the generation of efficient compiler-parallelized programs. In Marc Moonen and Francky Catthoor, editors, *Algorithms and Parallel VLSI Architectures, III*, pages 331–340, Amsterdam, 1995. Elsevier.

[VWD94] Hervé Le Verge, Doran K. Wilde, and Vincent Van Dongen. La synthèse de nids de boucles avec la bibliothèque polyédrique. In Luc Bougé, editor, *RenPar'6*. ENS Lyon, June 1994.

[WL91] M. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[Xue94] J. Xue. Automatic non-unimodular transformations of loop nests. *Parallel Computing*, 20(5):711–728, May 1994.

[ZBG88] H. P. Zima, H. J. Bast, and M. Gerndt. SUPERB : A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.