Supernode Partitioning

F. Irigoin
R. Triolet

Ecole Nationale Supérieure des Mines de Paris
Paris, France

Abstract

Supercompilers must reschedule computations defined by nested DO-loops in order to make an efficient use of supercomputer features (vector units, multiple elementary processors, cache memory, etc...). Many rescheduling techniques like loop interchange, loop strip-mining or rectangular partitioning have been described to speedup program execution. We present here a class of partitionings that encompasses previous techniques and provides enough flexibility to adapt code to multiprocessors with two levels of parallelism and two levels of memory.

1. Introduction

Supercomputer features, like vector units and vector registers, multiple elementary processors, cache and local memories, do not provide the performance level claimed by the manufacturers and expected by users on all kinds of programs. The necessary adaptation has always been first applied by hand before automatic tools were made available. Some practical knowledge has now be gained for new architectures,[14] with MIMD and vector capability plus a memory hierarchy, that shows the interest of DO loop blocking.

We present here a new program transformation, called **supernode partitioning**. The basic idea is to aggregate many do-loop iterations, called **nodes** so as to provide vector statements, parallel tasks and data reference locality when applied to perfectly nested DO loops.

In section 2 this transformation is applied to a simple example to show how it differs from other transformations and what can be expected from it. The next two sections are devoted to preliminaries. First assumptions on initial code are given and then bases of dependence analysis are briefly sketched. A new concept, called the **dependence cone**, is introduced as a basic tool for partitioning.

In section 5, supernode partitioning is studied. First four constraints that should reasonably be met by any partitioning are discussed. Hyperplane partitioning is introduced and a validity condition is derived from dependences. This is generalized to partitioning with multiple hyperplanes and other conditions on partitioning are introduced to meet the four initial constraints. Finally an opposite approach, based on node **clustering** with a basis to keep together computations sharing data, leads to the same result. It is shown to be dual of hyperplane partitioning. Validity conditions, that can be checked automatically with a dependence cone, are developed in both cases.

Once supernodes are defined it remains to generate code. This is the purpose of section 6. First of all a control structure must be decided to ensure a proper global scheduling of supernodes, compatible with dependences, and another one to schedule locally nodes inside supernodes. We show that linear schedulings provide in both cases one sequential outer loop and $N-1$ inner parallel loops, where N is the number of loops in the initial code.

Many supernode partitionings as well as global and local linear schedulings are valid and we show how they can be used to exploit various features of two dissimilar machines, the Cray-2 and the Alliant FX-8. The same set of criteria, ordered in two different ways, leads to two different partitionings for the same initial program.

Finally automatic code generation in 2-D is sketched. A general control frame is used and every parameter turns out to be the result of operations on linear systems of equalities and inequalities, because of assumptions made in part 4 and of the partitioning technique chosen.

2. Example

Consider program 1 which performs a relaxation step:

```
        DO I = 1, L
          DO J = 1, M
(S)         T(I,J) = (T(I-1,J)+T(I,J)+T(I+1,J)
                    +T(I,J-1)+T(I,J+1))*0.2
        ENDDO I, J
```

Program 1: A Five Point Relaxation Step

The relationship that exists between different iterations of statement S is depicted in figure 1, where each dot represents one iteration and each arrow a flow of data between two iterations.

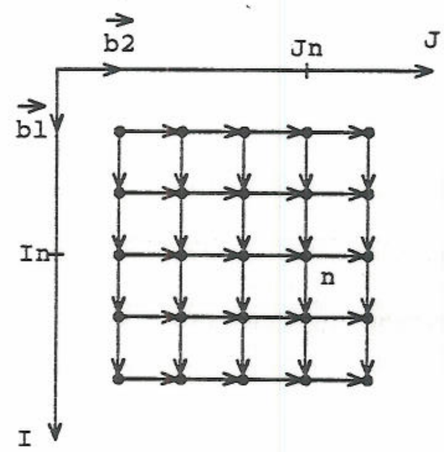

Figure 1: Dependences for Program 1

With this example usual program transformations like strip-mining[26] and loop interchange[2] do not provide any parallelism. The hyperplane method[18, 17, 5] should be applied to generate vector statements but commercial supercompilers do not apply it and keep loops I and J sequential. Furthermore, if the hyperplane method were available, it should be combined with strip-mining to use for example the 8 vector units of an Alliant FX-8.

But the resulting ordering does not allow data re-use, neither in registers nor in cache memory since a whole diagonal of array T must be computed before values produced as T(I,J) are used as T(I,J–1) and T(I–1,J) to compute the next diagonal. Moreover the synchronization overhead cannot be kept low on multiprogramming oriented machines like Cray-2, Cray-XMP or IBM-3090-VF since a partition size is bounded by a linear function of T's dimensions.

These conflicting goals are met with the partitioning of figure 2, where iterations of statement S are clustered into blocks that can be executed in parallel, front by front. Each block is labeled by its front number. The size of a block can be scaled up or down to adjust synchronization overhead versus the parallelism degree of fronts, i.e. the number of supernodes per front. Within a block, computations can be performed in vector mode along direction $\vec{p}_2$ and one vector register can be used three times along direction $\vec{p}_1$, saving two loads out of five.

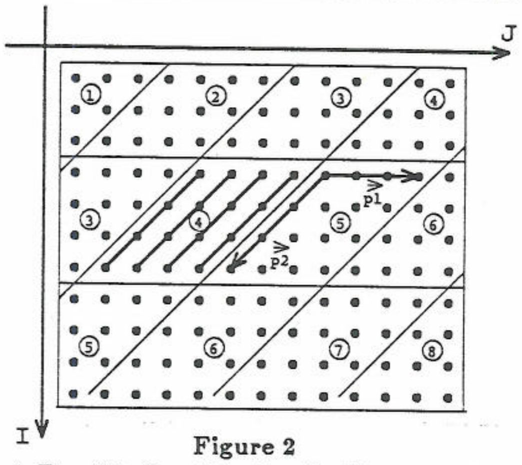

Figure 2
A Possible Partitioning for Program 1

Supernode partitioning is more general than rectangular partitionings[22]. It can be applied more often and provides more flexibility.

3. Terminology and Assumptions

Algorithms of interest can be written as sets of **perfectly** nested DO-loops, whose upper and lower bounds are all **linear**, enclosing a loop body S with linear array references. That is, each subscript expression must be a linear expression over the scalar integer variables of the program. The loop body must not contain any GOTO exiting one or many loops and all statements must be at the same nesting level. Loop increments are equal to 1 (*normalized* loops) and so the initial execution ordering is the lexicographic order on index values.

Such algorithms can also be expressed by a set of uniform recurrence equations[15] as used for systolic machines, but their domain must be bounded.

With the assumption on loop bound linearity, the sets of computations considered are finite convex polyhedra of some **iteration space** Z^N, where N is the number of nested loops and the dimension of the space. Each element, an iteration of loop body S, is called a **node** n and is referred to by its **iteration vector** $\vec{j}$. These vectors have the values of loop indices as coordinates in the initial basis $B = (\vec{b}_1, \vec{b}_2)$; in figure 1, node

n 's iteration vector is (I_n , J_n). A set of nodes is called a **supernode**. A few supernodes are shown on figure 2.

The set of computations is called the **computation domain**. It is defined by a set of **linear inequalities** derived from loop bounds. For example, program 1's computation domain is defined by the following system:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{pmatrix} \vec{j} \leq \begin{pmatrix} L \\ M \\ -1 \\ -1 \end{pmatrix} .$$

where $\vec{j}$ is expressed in the initial basis B .

Thus a computation domain is defined by a matrix A and a vector $\vec{a}$ and each iteration vector of this domain verifies:

$$A \, \vec{j} \leq \vec{a}$$

4. Dependences

Schedulings of nodes are constrained by different **dependence** relations[16] , whose most intuitive one is the **data flow dependence** (also called a **true dependence**): a node n_2, using some data produced by node n_1, depends on n_1 and cannot be executed concurrently to n_1. The set of nodes n_2 depending on n_1 is characterized by the **dependence vectors** $\vec{d} = \vec{j}_2 - \vec{j}_1$.

For each pair of array references that can induce such a dependence the set of possible values for $\vec{d}$ is usually upper approximated by another set that contains all $\vec{d}$'s and some extra elements. For instance Wolfe uses a **dependence direction vector**[26] which indicates if $\vec{d}$'s coordinates are less than, equal to or greater than 0.

In our case, this set is characterized by a convex polyhedron[17, 25, 12] D which can be automatically computed from systems of linear equalities and inequalities describing dependence conditions (equality of array references), belonging to the computation domain (deduced from A and $\vec{a}$) and definition of $\vec{d}$. These systems are projected on $\vec{d}$'s coordinates and finally, D is obtained by computing the convex hull of the projected systems. D is given by its **generating system**[24] : three sets of vertices, rays and lines.

This computation must be performed for each pair of references in the loop body and these elementary dependences must be combined to tell whether an iteration $\vec{j}_1$ depends directly or indirectly on an iteration $\vec{j}_2$ and must always be executed afterwards.

The transitive closure of all dependences provides this ordering relation between nodes. In our case the transitive closure is obtained by putting together the elementary generating systems and by transforming vertices into rays and each line into 2 rays. The final set of rays $R = (\vec{r}_1, \vec{r}_2, ..., \vec{r}_k)$ is called the **dependence cone**[12] , and provides the following equation:

$$\vec{j}_2 \textit{ must execute after } \vec{j}_1 \quad => $$
$$\exists \ \lambda_1, \lambda_2, ..., \lambda_k , \ \cdots \in N$$
$$\vec{d} = \vec{j}_2 - \vec{j}_1 = \sum_k \lambda_k \ \vec{r}_k$$

There is no equivalence because many approximations occur during the dependence cone computation and because the dependence distance vector set may not be a cone.

Similar dependence cones can be derived from a set of uniform recurrence equations used for systolic algorithm[21] , and less precise ones from Wolfe's dependence direction vectors (rays are parallel to basis vectors).

The relation *must execute after* is called *depends on* in the following and $\vec{j}_2$ depends on $\vec{j}_1$ is denoted by $\vec{j}_1 \underset{\delta}{<} \vec{j}_2$.

When the dependence vectors can be exactly computed, it is possible to decompose the computation domain in connected components that can be executed concurrently[23] , thus we can assume that the iteration space is connected. The cone dimension is supposed here to be N, the dimension of the iteration space, but it is shown in [13] how to handle other cases.

For program 1, the dependence cone is given by $\vec{r}_1 = \vec{b}_1$ and $\vec{r}_2 = \vec{b}_2$ which in this simple case could have been computed from the dependence direction vectors (=,<) and (<,=).

To see better how the dependence cone can summarize elementary dependences consider program 2 which is part of a Gaussian elimination for banded matrices[9] .

```
        DO I = 1, N
           DO J = 1, M
S:            Y(I+J) = F(Y(I), Y(I+J))
           ENDDO
        ENDDO
```

Program 2
Gaussian Elimination of Banded Matrices

F is a side-effect free function and two pairs of references must be tested: (Y(I), Y(I+J)) and (Y(I+J),Y(I+J)). There are no constant dependence distance vectors since a monodimensional array is used at a nesting level of 2. Linear systems built for these two pairs provide the following information on dependence distance vectors by projection on the dependence space:

$$\Pi_{1,2} = \begin{cases} d_i \geq 1 \\ d_i + d_j \geq 1 \end{cases}$$

$$\Pi_{1,3} = \begin{cases} d_i + d_j = 0 \end{cases}$$

These polyhedra are reduced to the smallest polyhedra containing their lexico-positive part and are expressed as generating systems (see [12] for more details):

$$\Pi_{1,2} = \left(\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}, \emptyset \right)$$

$$\Pi_{1,3} = \left(\left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right\}, \left\{ \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right\}, \emptyset \right)$$

These two elementary dependence relations are unioned and transitively closed into a new relation generated by:

$$R = \left(\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\} \right)$$

By redundance elimination this first generating system is reduced to:

$$R = \left(\begin{pmatrix} 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right)$$

This let us find possible supernode partitioning while Wolfe's combined dependence direction vector for the same loop is $(<, *)$ and contains a line. Dependence cone R_W

$$R_W = \left(\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\} \right)$$

would be derived from it.

The subsets of the dependence space defined by R and R_W are depicted in figure 3.a and 3.b. It is obvious that dependence cone R is more accurate than R_W.

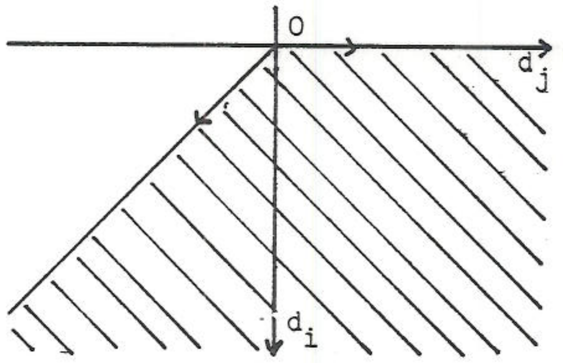

Figure 3.a: Dependence Set Generated by R

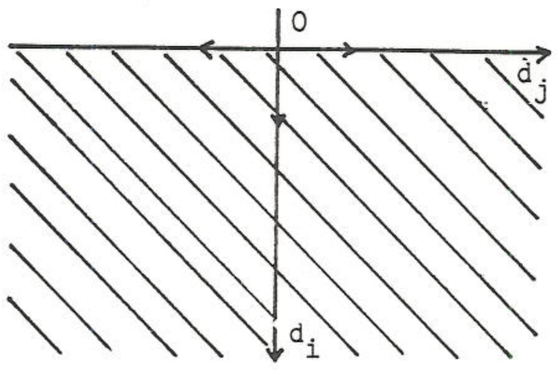

Figure 3.b: Dependence Set Generated by R_W

Dependence cones defined by generating systems provide the kind of information that was previously available only for constant dependence distance vectors. However rays provide only a dependence direction and not a distance. Some useful information is lost for the minimum distance partitioning[23].

5. Partitioning

5.1. Constraints on Partitioning

Four constraints are imposed on supernodes. They must be atomic, identical by translation, bounded, and they must tile the computation domain.

Each supernode must define an **atomic** task, that can be executed without synchronization once it is started. All necessary inputs must be available at the beginning and all outputs are made available to other supernodes at the end. In other words, all synchronization points are beginnings or ends of supernodes. This constraint implies the following condition on supernodes:

$$\forall \vec{j}_1, \vec{j}_1{}' \in s_1 \quad \forall \vec{j}_2, \vec{j}_2{}' \in s_2 \neq s_1 \qquad (1)$$

$$\vec{j}_1 \underset{\delta}{<} \vec{j}_2 \;=>\; \vec{j}_1{}' \underset{\delta}{<} \vec{j}_2{}' \quad \textit{or} \quad \vec{j}_1{}' \underset{\delta}{<>} \vec{j}_2{}'$$

i.e.

$$\vec{j}_1 \underset{\delta}{<} \vec{j}_2 \;=>\; \textit{not} \; (\vec{j}_2{}' \underset{\delta}{<} \vec{j}_1{}' \;)$$

where $\underset{\delta}{<>}$ means *are not dependent*. The partitioning relation must be compatible with the partial order on nodes, which means that a partial order on supernodes is induced and that supernodes can be scheduled atomically without violating dependences.

The constraint of **identity** is imposed to keep code generation simple enough to be automated: each supernode must be the image of any other one by a translation, except when it crosses the computation domain boundaries (for instance see the unique supernode of front 8 in figure 2). In the later case, supernodes are called **partial supernodes**, in opposition to full

supernodes.

Supernodes must be **bounded** to make sure the array regions they refer can be adapted to the high speed memory (register, cache or local).

And finally supernodes must **tile** exactly the iteration space to make sure each iteration of S is executed once and only once.

5.2. Hyperplane Partitioning

Let $\vec{h}$ be a vector of $\mathbf{Q}^N$, where $\mathbf{Q}$ is the set of rational numbers, and consider the following partitioning relation:

$$\vec{j}_1 \in s \text{ and } \vec{j}_2 \in s \iff \lfloor \vec{h}.\vec{j}_1 \rfloor = \lfloor \vec{h}.\vec{j}_2 \rfloor \qquad (2)$$

It slices the iteration space with regularly spaced hyperplanes defined by:

$$\vec{h}\ \vec{j} = K \qquad K \in \mathrm{N}$$

This can be seen on figure 4.a. Hyperplane direction vector $\vec{h}_1$ is not at scale to be visible. Its coordinates are (1/2, -1/4).

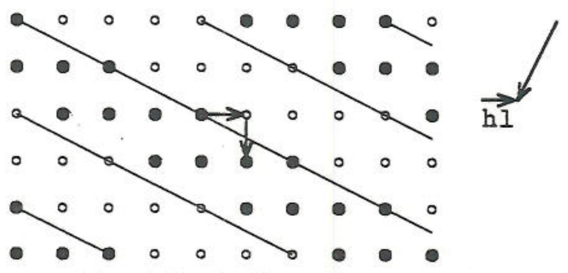

Figure 4.a: Partitioning With One Hyperplane

The following theorem shows that condition (1) is satisfied if $\vec{h}^T\ R \geq \vec{0}$. Intuitively this latter condition means that all rays, and as a consequence, all dependence vectors, cross any hyperplane defined by $\vec{h}$ in the same direction, from one supernode to another. The opposite condition, $\vec{h}^T\ R \leq \vec{0}$, could also be used and would define an opposite set of directions. We choose the former condition:

$$\vec{h}^T\ R \geq \vec{0} \qquad (3)$$

so that $\vec{h}$ gives the ray crossing direction.

Theorem 1

Condition (3) is a sufficient condition for condition (1).

Proof

Let's suppose (1) is not true while (3) holds:

$$\exists\ \vec{j}_1, \vec{j}_1{}' \in s_1 \quad \exists\ \vec{j}_2, \vec{j}_2{}' \in s_2 \neq s_1$$

$$\vec{j}_1 \underset{\delta}{\leq} \vec{j}_2 \text{ and } \vec{j}_2{}' \underset{\delta}{\leq} \vec{j}_1{}'$$

This can be rewritten:

$$\vec{j}_2 = \vec{j}_1 + \sum_k \lambda_k \vec{r}_k \qquad \lambda_k \geq 0$$

$$\vec{j}_1{}' = \vec{j}_2{}' + \sum_k \mu_k \vec{r}_k \qquad \mu_k \geq 0$$

These two equations and condition (3) imply:

$$\vec{h}\ .\ \vec{j}_2 \geq \vec{h}\ .\ \vec{j}_1 \text{ and } \vec{h}\ .\ \vec{j}_1{}' \geq \vec{h}\ .\ \vec{j}_2{}'$$

because λ_k 's and μ_k 's are positive and thus:

$$\lfloor \vec{h}\ .\ \vec{j}_2 \rfloor \geq \lfloor \vec{h}\ .\ \vec{j}_1 \rfloor \text{ and } \lfloor \vec{h}\ .\ \vec{j}_1{}' \rfloor \geq \lfloor \vec{h}\ .\ \vec{j}_2{}' \rfloor$$

With definition (2) of hyperplane partitioning, this implies:

$$\lfloor \vec{h}\ .\ \vec{j}_2 \rfloor = \lfloor \vec{h}\ .\ \vec{j}_1 \rfloor = \lfloor \vec{h}\ .\ \vec{j}_1{}' \rfloor = \lfloor \vec{h}\ .\ \vec{j}_2{}' \rfloor$$

and all nodes must belong to the same supernode, which is inconsistent with the hypothesis $s_2 \neq s_1$.

End of Proof

If we were dealing with real or rational numbers, condition (3) would also be a **necessary** condition of (1). This is not the case. Condition (3) is a bit too strong and some valid partitionings as obtained by a combination of wavefronting and strip-mining do not meet it[13] .

Vectors $\vec{h}$ that meet condition (3) are called **valid** hyperplanes. This condition defines the opposite of R 's **polar cone**[24] , $\overline{R}^*$. Thus each valid hyperplane is a positive linear combination of $\overline{R}^*$'s rays. The dimension of $\overline{R}^*$ depends on R and, for instance, is 0 when the program is fully sequential. The partitioning method presented here apply when $\overline{R}^*$ and R are full-dimensional.

$\overline{R}^*$ may not be full-dimensional when there are too many dependences. In this case supernode partitioning should be applied to a subset of the initial nested loops. Remaining loops are kept sequential to remove some dependences.

$\overline{R}^*$ may also not be full-dimensional when there are not enough dependences, as in the matrix product. One solution is to isolate fully parallel loops and to apply supernode partitioning to the remaining ones. Parallelism is then available but data locality and synchronization overhead cannot be controlled. Another solution is to introduce pseudo-dependences in R and $\overline{R}^*$ to define the partitioning. These pseudo-dependences are ignored when computing possible schedulings.

5.3. Generalized Hyperplane Partitioning

This hyperplane partitioning can be generalized by using a set H of **valid** hyperplanes $\vec{h}_j$. The partitioning relation becomes:

$$\vec{j}_1 \in s \text{ and } \vec{j}_2 \in s \iff \qquad (4)$$

$$\forall j \ \lfloor \vec{h}_j\ .\ \vec{j}_1 \rfloor = \lfloor \vec{h}_j\ .\ \vec{j}_2 \rfloor$$

The validity condition becomes $H^T\ R \geq 0$, where 0 denotes a null matrix with proper dimensions. It can be easily shown to be a sufficient condition for (1) since two supernodes are separated by at least one hyperplane and since the previous proof can be applied to this hyperplane.

Figure 4.a and 4.b show two hyperplane partitionings for program 1. The partitioning defined by $\vec{h}_1$ is not valid because $\vec{h}_1$ does not belong to $\vec{R}^*$, which here is equal to R (see figure 1 and §4). Although $\vec{h}_2$ is a valid hyperplane, the partitioning generated by $(\vec{h}_1, \vec{h}_2)$ leads to unschedulable supernodes: A must be executed before B, and B before A. A valid partitioning was shown in figure 2.

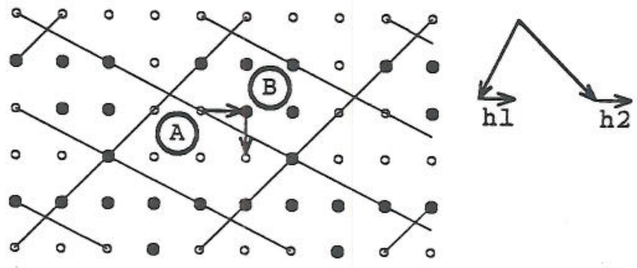

Figure 4.b: Partitioning With Two Hyperplanes

5.4. Identity Condition

Theorem 2

Supernodes are equal up to a translation iff H is free.

Proof

We must show that for any two supernodes s_1 and s_2 there exists at least one translation $\vec{t}$ that maps any node of s_1 onto a node of s_2. Let k_1^j and k_2^j be the integer coordinates of s_1 and s_2:

$$\forall \vec{j} \in s_1 \quad \forall \vec{h}_j \in H \qquad \lfloor \vec{h}_j \cdot \vec{j} \rfloor = k_1^j$$
$$\forall \vec{j} \in s_2 \quad \forall \vec{h}_j \in H \qquad \lfloor \vec{h}_j \cdot \vec{j} \rfloor = k_2^j$$

Translation $\vec{t}$ is such that:

$$\forall \vec{h}_j \in H \quad \forall \vec{j}_1 \quad \lfloor \vec{h}_j \cdot \vec{j}_1 \rfloor = k_1^j \quad => $$
$$\lfloor \vec{h}_j \cdot (\vec{j}_1 + \vec{t}) \rfloor = k_2^j$$

Let r_1^j, r_2^j and r_t^j be the rational parts of previous dot products. The condition on $\vec{t}$ can be rewritten:

$$\begin{cases} \vec{h}_j \cdot \vec{j}_1 = k_1^j + r_1^j \\ \vec{h}_j \cdot (\vec{j}_1 + \vec{t}) = k_1^j + r_1^j + k_t^j + r_t^j = k_2^j \end{cases}$$

This holds for any $\vec{j}_1$, thus $r_1^j \in [0..1[$. To keep $\vec{j}_1 + \vec{t}$ in s_2, r_t^j must be equal to 0. Combining the two previous equalities and using a matrix notation, the condition on $\vec{t}$ becomes:

$$H^T \vec{t} = \vec{k}_2 - \vec{k}_1$$

where k_1^j's and k_2^j's are the coordinates of $\vec{k}_1$ and $\vec{k}_2$. This system has solutions if H is free. The solution is unique when H is a generator, i.e. when its rank is equal to the iteration space dimension N.

End of Proof

In figure 4.c, a third partitioning direction $\vec{h}_3$ is used to show what hapens when too many hyperplanes are used. H is no longer square and cannot be inverted. Obviously supernodes are not equal.

These solutions are integer if H^{-1} has integer components. This is another important condition since we have to map integer point onto integer point to generate the same code for each supernode. This last condition is not met by partitioning of figure 4.b: nodes cannot always be mapped onto nodes from another supernode with an integer translation. Let's compute H^{-1}:

$$\vec{h}_1 = \begin{pmatrix} 1/2 \\ -1/4 \end{pmatrix} \qquad \vec{h}_1 = \begin{pmatrix} 1/4 \\ 1/4 \end{pmatrix} \qquad H = \begin{pmatrix} 1/2 & 1/4 \\ -1/4 & 1/4 \end{pmatrix}$$

$$H^{-1} = \frac{16}{3} \begin{pmatrix} 1/4 & 1/4 \\ -1/4 & 1/2 \end{pmatrix}$$

For most $\vec{k}_1$ and $\vec{k}_2$, $\vec{t} = H^{-1}(\vec{k}_2 - \vec{k}_1)$ will not have integer coördinates.

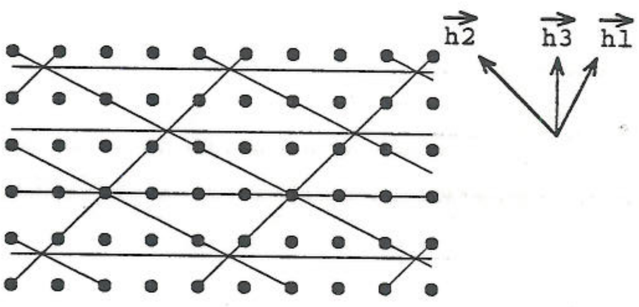

Figure 4.c: Partitioning With Three Hyperplanes

5.5. Finite Supernodes

The number of nodes (i.e. integer points) of a supernode is finite if the supernode is bounded.

Theorem 3

Supernodes are bounded if and only if H is a generator of the iteration space.

Proof

• It is a necessary condition. If H were not a generator of the iteration space, there would exist a non-empty orthogonal subspace $H^\perp$. Any **integer** vector $\vec{k}^\perp$ of $H^\perp$ verifies:

$$\forall \vec{h}_j \in H \quad \vec{h}_j \cdot \vec{k}^\perp = 0$$

and any supernode s containing a node $\vec{j}$ contains also the infinite set of nodes

$$\{ \vec{j}' \ / \exists\ \lambda \in \mathbf{Z} \ s.t.\ \vec{j}' = \vec{j} + \lambda\, \vec{h}^{\perp} \ \}$$

since:

$$\forall \vec{h}_j \in H \quad \vec{h}_j \,.\, \vec{j}' = \vec{h}_j \,.\, \vec{j} + \vec{h}_j \,.\, \vec{h}^{\perp}$$
$$= \vec{h}_j \,.\, \vec{j}$$

Vector $\vec{h}^{\perp}$ exists because H is a set of **rational** vectors.

• It is a sufficient condition. A polyhedron that is not bounded contains rays. Let $\vec{v}$ be one such ray. By definition of supernodes and rays we get:

$$\forall \lambda > 0 \quad k_j \le \vec{h}_j \,.\, (\vec{j} + \lambda \vec{v}) < k_j + 1$$

This implies $\vec{h}_j . \vec{v} = 0$ for all $\vec{h}_j$. As H is a generator, it is possible to build a basis H' from H and to use the corresponding subset of equations $(Hprime)^T \vec{v} = \vec{0}$ to show $\vec{v} = \vec{0}$.

End of Proof

For instance figure 4.a's partitioning is unbounded.

5.6. Proper Partitioning

The tiling condition is satisfied by definitions (2) and (4) that maps any node into exactly one supernode.

The other three conditions on partitioning presented in section 5.1 are met if H is a **basis** whose vectors define valid hyperplane partitionings, i.e. belong to $\overline{R}^*$, the opposite of the dependence polar cone. It is possible to build such a basis when R (and hence $\overline{R}^*$) is full-dimensional.

5.7. Basis Clustering

The basic idea is to put together nodes that depend on each other to propagate values from producer to consumer **inside** one supernode and to limit communications between supernodes. This idea, depicted on figure 5, leads to the following definition:

$$\vec{j}_1 \in s \quad \text{and} \quad \vec{j}_2 \in s \quad <=> \quad \lfloor \vec{j}_1^{\,P} \rfloor = \lfloor \vec{j}_2^{\,P} \rfloor \qquad (5)$$

where $\vec{j}^{\,P}$ denotes the coordinates of $\vec{j}$ in some **clustering basis** P and where the floor function and the equality are applied componentwise.

Supernodes are equal to unit cells of basis P. This gives a linear condition for a node $\vec{j}$ to belong to a supernode of origin $\vec{j}_0$:

$$\vec{0} \le P^{-1}(\vec{j} - \vec{j}_0) < \vec{1}$$

where $\vec{j}$ and $\vec{j}_0$ are expressed in the initial basis B and $\vec{1}$ in P. To use integer linear algorithm, each term is multiplied by $|\,det(P)\,|$ and the strict inequality is removed:

$$\vec{0} \le |\,det(P)\,|\ P^{-1}(\vec{j} - \vec{j}_0) \le (\,|\,det(P)\,| - 1)\,\vec{1}$$

Since definitions (4) and (5) can be rewritten with matrix notation as:

$$\lfloor H^T \vec{j}_1 \rfloor = \lfloor H^T \vec{j}_2 \rfloor$$

and:

$$\lfloor P^{-1} \vec{j}_1 \rfloor = \lfloor P^{-1} \vec{j}_2 \rfloor$$

(with $\vec{j}_1$ and $\vec{j}_2$ expressed in the initial basis B) it is clear that each clustering basis $P = (..., \vec{p}_i, ...)$ is related to a partitioning basis $H = (..., \vec{h}_j, ...)$ by the relation $H^T = P^{-1}$ which can be rewritten $\vec{h}_j . \vec{p}_i = \delta_{i,j}$ where $\delta_{i,j}$ is the Kronecker function. This means that the $\vec{h}_j$'s are orthogonal to P faces and the same for the $\vec{p}_i$'s relatively to H.

In order to meet condition (1), a clustering basis P must satisfy the condition $P^{-1}R \ge 0$ since it defines the same partitioning than $H = (P^{-1})^T$ and since this partitioning is valid if and only if $H^T\ R \ge 0$. This new condition means that all dependence vectors must have positive coordinates in P. Moreover, the matrix P must have integer components in the initial basis to generate identical **integer** supernodes. This condition is easier to understand than the condition on H^{-1} developed for theorem 2.

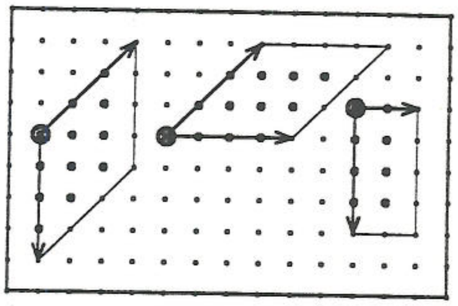

Figure 5
Three Valid Clustering Bases
for Program 1

6. Code Generation

6.1. Generation Parameters

Once H (or P) are chosen many schedulings of nodes and supernodes are possible. To keep control code generation simple and to produce efficient code linear schedulings are chosen. A linear scheduling is defined by one vector $\vec{\sigma}$:

Definition: Let $\vec{j}_1$ and $\vec{j}_2$ be two iteration vectors. $\vec{j}_2$ is executed after $\vec{j}_1$ with linear scheduling $\vec{\sigma}$ iff $\vec{\sigma}.\vec{j}_1 < \vec{\sigma}.\vec{j}_2$

These linear schedulings are also used as temporal functions to generate systolic arrays from systolic equations. With Fortran control structures a linear scheduling given by $\vec{\sigma}$ is obtained with one sequential loop along direction $\vec{\sigma}$ enclosing $N-1$ parallel loops. These loops are chosen to scan the whole iteration space, and the parallel ones are characterized by basis vector of the subspace orthogonal to $\vec{\sigma}$.

Definition: A linear scheduling compatible with dependences is called a valid linear scheduling.

Theorem 4

The set of valid linear schedulings is defined by $\vec{\sigma}^T R \geq \vec{1}^T$.

Proof

Let $\vec{j}_1$ and $\vec{j}_2$ be iteration vectors of two dependent nodes:

$$\vec{j}_1 \underset{\delta}{<} \vec{j}_2 => \vec{j}_2 = \vec{j}_1 + \sum_k \lambda_k \vec{r}_k$$

$$\forall k \;\; \lambda_k \geq 0 \text{ and } \exists \; j \;\; s.t. \;\; \lambda_j > 0$$

This implies:

$$\vec{\sigma} \,.\, \vec{j}_2 = \vec{\sigma} \,.\, \vec{j}_1 + \sum_k \lambda_k \, \vec{\sigma} \,.\, \vec{r}_k$$

Since $\vec{j}_2$ must execute after $\vec{j}_1$, $\vec{\sigma}.\vec{j}_2 > \vec{\sigma}.\vec{j}_1$ and $\Sigma \lambda_k \, \vec{\sigma}.\vec{r}_k > 0$. This must hold for any set of λ_k's as defined above and implies $\vec{\sigma}^T R > \vec{0}$. Since $\vec{\sigma}$ has rational coordinate and since $\lambda \, \vec{\sigma}$ and $\vec{\sigma}$ define the same linear ordering, the validity condition can be written:

$$\vec{\sigma}^T R \geq \vec{1}$$

End of Proof

Thus valid linear schedulings belong to a subset of valid partitioning hyperplanes. This subset is a polyhedron that can be defined by a generating system. Potential schedulings are positive linear combinations of its rays.

This analysis can be applied at the node level (and is better known as the hyperplane method) and at the supernode level since supernodes can be ordered by only one node (see condition (1)), their **origin** for example. A supernode origin is its node with minimal coordinates in P. These coordinates are equal by definition to the supernode coordinates.

Let $\vec{1}_H$ be the vector whose coordinates in basis H are all equal to 1. i.e. $\vec{1}_H = \Sigma \, \vec{h}_j$.

Theorem 5:

$\vec{1}_H$ is a valid linear scheduling.

Proof

We must show $\vec{1}_H^T R > \vec{0}$ knowing that $H^T R \geq 0$ (see §5.4) and hence that $\vec{1}_H^T R \geq \vec{0}$. Suppose $\vec{1}_H^T R = \vec{0}$. By definition of $\vec{1}_H$ and since H is a valid partitioning:

$$\sum_j \vec{h}_j^T R = \vec{0} \text{ and } \forall \, j \;\; \vec{h}_j^T R \geq 0$$

This implies $H^T \; R = 0$. Since H is a basis, H^T can be inverted and $R = 0$ which is impossible for a dependence cone. 0 denotes a null matrix when convenient.

End of Proof

This shows that, under the full-dimensionality assumption on the dependence cone R, one outer sequential loop and $N-1$ inner parallel loops preserve dependences between supernodes and another similar set of loops preserve dependences between nodes. $\vec{1}_H$ is used to show the existence of parallel loops but many other valid global and local linear scheduling can be chosen. The global and local scheduling directions can be different.

Generation bases G, also called scanning bases, can be derived from any linear scheduling $\vec{\sigma}$. The $N-1$ parallel loops are defined by $N-1$ vectors of G that are orthogonal to $\vec{\sigma}$. The last vector of G defines the sequential direction. It has to be free with the others and to be oriented along $\vec{\sigma}$. Vector $\vec{\sigma}$ can sometimes be used. G must be unimodular (i.e. $| \, det(G) \, | = 1$) to map nodes (i.e. integer points) of the iteration space onto nodes of the initial one. For supernodes, the same condition must hold for the change of basis matrix from G to P.

Thus, once a partitioning is defined by a matrix H, a global and a local iteration directions as well as the corresponding generation bases have still to be defined to scan supernodes inside the computation domain and to scan nodes inside each supernode. The supernode size can easily be adjusted by an integer scaling factor applied to H or P without changing the partitioning and iteration directions. The partitioning grid can also be translated and so a partitioning origin must be chosen.

6.2. Choosing a Partitioning

Supercomputer architectures are too intricate to derive an analytical expression for the partitioned program execution time. Nevertheless several key factors are well known. Vector statements should be generated; vectors should have a constant length; data stored in vector registers should be re-used; data accessed many times should be kept in cache; memory references should be done with stride 1; the workload should be balanced between elementary processors; synchronization overhead should be minimized; and so on. Supernode partitioning is flexible enough (partitioning directions, local iteration direction, supernode size, ...) to fulfill any of these factors. But these goals are conflicting and priorities must be given to each of them according to the target architecture.

Partitioning on figure 2 was designed for a 4-processor Cray-2. A hyperplane direction was chosen to get vectors of constant length (64: the vector register length[†]). The second partitioning direction was chosen to save two loads out of five with the references T(I,J–1), T(I,J), T(I, J+1). Another direction could have been chosen with references T(I–1,J), T(I,J), T(I+1,J). Vector access stride cannot be 1. Load imbalance was considered second to vector performances and synchroni-

[†]Partitionings depicted on figure 2 and 6 have been scaled down to keep nodes visible.

zation overhead can be kept low with large supernodes, along direction $\vec{p}_2$.

Another partitioning is presented in figure 6 for an Alliant FX-8[10] . As potential speedup is 2 to 3 with vector statements and 8 with code spreading on elementary processors, we decided to promote parallelization versus vectorization. This partitioning was chosen because it provides fronts of 8 supernodes or more before the previous one. Moreover a parallelism degree of 8 is quickly reached because supernode size can be small due to an efficient synchronization hardware. This partitioning drawback is the **variable length** of vector statements generated to exploit vector units (see figure 6). Vector registers are not re-used since the cache memory has the same throughput.

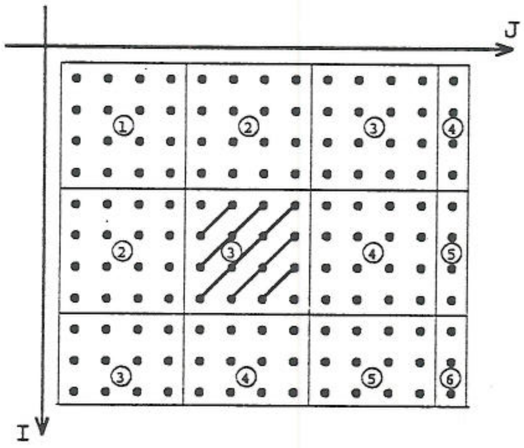

Figure 6
A Partitioning for Program 1
on Alliant FX-8

6.3. Automatic Code Generation

Once a partitioning and the other code generation parameters have been chosen, it is possible to check automatically their validity with the dependence cone and to compute the partitioned code. In 2-D, we use the control structure given in program 3.

```
DOSEQ L1 = LB_L1, UB_L1
   DOALL L2 = MAX(LB_L2₁, LB_L2₂, ...),
             MIN(UB_L2₁, UB_L2₂, ...)
      IF (T1) THEN
         DOSEQ l1 = lb_l1, ub_l1
            DOVEC l2 = MAX(lb_l2₁, lb_l2₂, ...),
                      MIN(ub_l2₁, ub_l2₂, ...)
               loop body S with new subscripts
      ELSE IF (T2) THEN
         DOSEQ l1 = lb_l1', ub_l1'
            DOVEC l2 = MAX(lb_l2₁', lb_l2₂', ...),
                      MIN(ub_l2₁', ub_l2₂', ...)
               loop body S with new subscripts
      ELSE IF (...) THEN
         ...
```

Program 3
Skeleton for Automatic Code Generation (2-D)

Fronts of supernodes are scanned by one outer sequential loop (L1) along dependences (supernodes in figure 2 and 6 are labeled by L1 values, the front number) and every front by a parallel inner one (L2) to use the elementary processors. Tests (T1, T2, ...) are applied to check whether (and how) each supernode crosses the computation domain boundaries. Nodes are scanned inside each supernode by a sequential and a vector loops (l1 and l2). Finally a change of basis is applied to subscript expressions as can be seen in the code fragment of program 4, where K is the supernode size and (I0, J0) its origin coordinates. The transformation linearity keeps array subscript expressions linear, and constant stride memory references are still available for vector code generation.

```
    DOSEQ l1 = 0, K-1
      DOVEC l2 = 0, K-1
(S)     T(I0+l2,J0+l1-l2) = (T(I0+l2-1,J0+l1-l2)
   &       +T(I0+l2,J0+l1-l2)+T(I0+l2+1,J0+l1-l2)
   &       +T(I0+l2,J0+l1-l2-1)+T(I0+l2,J0+l1-l2+1))*0.2
```

Program 4
Code Fragment for Full Supernodes of Figure 2

New loop boundaries and tests are computed with well known integer linear system algorithms, like testing feasibility[8] , projection, redundance elimination[4, 3] , etc... More details can be found in [13].

To give a flavor of the techniques used, let see how bounds for loop L1 and L2 can be computed. Supernodes (L1,L2) of interest contain at least one node (I,J) of the initial computation domain. Let (P1,P2) be the origin of the partitioning and (I0,J0) the origin of supernode (L1,L2). L1 and L2 are coordinates in the generation basis G, while the other coordinates are relative to the initial basis B. The partitioning is defined here by the clustering basis P. The previous sentences can be mathematically rewritten:

$$\begin{pmatrix} I0 \\ J0 \end{pmatrix} = \begin{pmatrix} P1 \\ P2 \end{pmatrix} + G \begin{pmatrix} L1 \\ L2 \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \le |\, det(P)\,|\ P^{-1} \begin{pmatrix} I-I0 \\ J-J0 \end{pmatrix} \le (\,|\, det(P)\,| -1) \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$A \begin{pmatrix} I \\ J \end{pmatrix} \le \vec{a}$$

Lower and upper bounds for L1, LB_L1 and UB_L1, are obtained by projecting this system on L1 and by eliminating redundant constraints. This is feasible even when $\vec{a}$ contains variables as in §3. Then the new pair of inequalities on L1 is added to the initial system and this redundant system is projected on L1 and L2. Redundancy elimination is performed and multiple constraints, if any, are handled with MAX and MIN operators.

The same technique is applied to generate tests T1, T2, etc... because interesting predicates can always be expressed as linear conditions. It is also applied to generate bounds for loops l1 and l2. Thus it can be used to generate loop bounds for the hyperplane method or for loop interchanging[27] whatever the initial (linear) loop bounds are.

Conclusion

We have presented a new restructuring method, called supernode partitioning. This method provides enough parameters to adapt perfectly nested DO loops to multiprocessors with vector units and a two-level memory hierarchy. It can also be used for simple vector processors.

This program transformation can be implemented in supercompilers using the classical Banerjee-Wolfe dependence test like KAP[6], PFC[1] or VATIL[19], but would benefit from more sophisticated tests as presented in [17] and [25] and from the new concept of dependence cone[12]. Hyperplane partitioning can also be used to map systolic programs on fixed size arrays.[7, 20]

Algorithms have been developed to automate its application to 2-level nested loops. Work is underway to extend it to any depth and to non-perfectly nested loops. These algorithms can be used with simpler methods like loop interchange to compute new loop bounds.

Preliminary results on an Alliant FX-8 have shown a 50 % speed increase due to reference locality improvement[11]. We are also working on an IBM 3090 VF to study cache policy effects and on a Cray-2 to measure the effect of register re-use and to assess synchronization overhead. These experiments should lead to a better knowledge of key architectural parameters and to the development of a partitioning choice algorithm.

References

1. J. R. Allen and K. Kennedy, "PFC: a Program to Convert Fortran to a Parallel Form," in *Supercomputers, Design and Application*, ed. K. Hwang (1982). COMPSAC, Tutorial

2. J. R. Allen and K. Kennedy, "Automatic Loop Interchange," *SIGPLAN'84 Symposium on Compiler Construction*, pp.233-246 (June 1984).

3. C. Ancourt, "Utilisation de systèmes linéaires sur Z pour la parallélisation de programmes," ENSMP-CAI-87-E87, Ecole des Mines de Paris, Fontainebleau (France) (1987).

4. P. Cousot and N. Halbwachs, "Automatic Discovery of Linear Restraints Among Variables of a Program," *Conference Record of the Tenth Annual Symposium on Principles of Programming Languages* (1978).

5. R. G. Cytron, "Compile-Time Scheduling and Optimization for Asynchronous Machines," PhD Thesis, Report No. UIUCDCS-R-84-1739, University of Illinois at Urbana-Champaign (1984).

6. J. Davies, C. Huson, T. Macke, M. Wolfe, and B. Leasure, "The KAP/S-1: An Advanced Source-to-Source Vectorizer for the S-1 Mark IIa Supercomputer," *Int'l Conference on Parallel Processing*, pp.833-835 (Aug. 1986).

7. V. Dornic, "Méthodes de partitionnement des réseaux systoliques appliquées à la triangularisation de matrice," Rapport de DEA, IRISA, Rennes (Juin 1987).

8. R. J. Duffin, "On Fourier's Analysis of Linear Inequality Systems," *Mathematical Programming Study 1*, North-Holland (1974).

9. D. Gannon, "Restructuring Nested Loops on the Alliant Cedar Cluster: A Case Study of Gaussian Elimination of Banded Matrices," CSRD document no. 543, University of Illinois at Urbana Champaign (Feb. 1986).

10. R. Gottlieb, K. Kimball, T. Jaskiewicz, and R. Swift, "A New Way To Speed Up a Supercomputer," *Electronics* 58(30), pp.56-58 (July 1985).

11. F. Irigoin and R. Triolet, "Supernodes and Alliant FX/8 Minisupercomputer," ENSMP-CAI-86-081, Ecole des Mines de Paris, Fontainebleau (France) (Aug. 1986).

12. F. Irigoin and R. Triolet, "Computing Dependence Direction Vectors and Dependence Cones With Linear Systems," ENSMP-CAI-87-E94, Ecole des Mines de Paris, Fontainebleau (France) (1987).

13. F. Irigoin, "Partitionnement des boucles imbriquées : une technique d'optimisation pour les programmes scientifiques," Thèse de Doctorat d'Université, Université PARIS-VI, PARIS (1987).

14. W. Jalby and U. Meier, "Optimizing Matrix Operation on a Parallel Multiprocessor with a Hierarchical Memory System," *1986 Int'l Conference on Parallel Processing*, pp.429-432 (Aug. 1986).

15. R. M. Karp, R. E. Miller, and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *Journal of the ACM* 14(3), pp.563-590 (July 1967).

16. D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," *ACM Symposium on Principles of Programming Languages*, pp.207-218 (Jan. 1981).

17. R. H. Kuhn, "Optimization and Interconnection Complexity for: Parallel Processors, Single-stage Networks, and Decision Trees," Ph.D. Thesis, No. UIUCDCS-R-80-1722, University of Illinois at Urbana-Champaign (1980).

18. L. Lamport, "The Parallel Execution of DO Loops," *Communications of the ACM* 17(2), pp.83-93 (Feb. 1974).

19. A. Lichnewsky and F. Thomasset, "Techniques de Base sur l'Exploitation Automatique du Parallélisme dans les Programmes," *Calcul Parallèle à usage scientifique* (Oct. 1985).

20. D. I. Moldovan and J. A. E. Fortes, "Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays," *IEEE Transactions on Computers* 35(1), pp.1-12 (Jan. 1986).

21. C. Mongenet, "Une méthode de conception d'algorithmes systoliques. Résultats théoriques et réalisation.," Thèse de Doctorat, INPL, Nancy (1985).

22. J.-K. Peir, "Program Partitioning and Synchronization on Multiprocessor Systems," PhD Thesis, Report UIUCDCS-R-86-1259, University of Illinois at Urbana-Champaign (March 1986).

23. J.-K. Peir and R. Cytron, "Minimum Distance: A Method for Partitioning Recurrences for Multiprocessors," *1987 Int'l Conference on Parallel Processing*, pp.217-225 (Aug. 1987).

24. A. Schrijver, *Theory of Linear and Integer Programming*, Wiley, Chichester (1986).

25. R. Triolet, F. Irigoin, and P. Feautrier, "Direct Parallelization of CALL Statements," *SIGPLAN'86 Symposium on Compiler Construction*, pp.176-185 (June 1986).

26. M. J. Wolfe, "Optimizing Supercompilers for Supercomputers," PhD. Thesis, Report No. UIUCDCS-R-82-1105, University of Illinois at Urbana-Champaign (1982).

27. M. J. Wolfe, "Advanced Loop Interchanging," *Int'l Conference on Parallel Processing*, pp.536-543 (Aug. 1986).