Massive Parallelism through Program Restructuring

Michael Wolfe

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 90-009

June, 1990

Massive Parallelism through Program Restructuring

Michael Wolfe
Oregon Graduate Institute of Science and Engineering
19600 NW von Neumann Drive
Beaverton, OR 97006

mwolfe@cse.ogi.edu
503-690-1153

Abstract

A technique for mapping algorithms to massively parallel processors is described; this method differs from previous work by focusing on explicit program restructuring as opposed to manual or algebraic mapping, and allows nonlinear as well as linear mappings. This method benefits from previous work in program restructuring and systolic array synthesis and thus will be simple to implement.

Prologue

Systolic array mapping of a nested-loop iterative algorithm has been posed as finding a linear mapping (R) of the index set of the algorithm ($\mathbf{I} \subset Z^n$) such that certain constraints are met. The algorithm is usually restricted to tightly nested loops (n nested loops) with invariant or perhaps triangular loop limits that define a finite index set or *iteration space*. The body of the algorithm computes new values for one or more array elements from previously computed values and initial conditions. The mapping R is usually defined as $\begin{pmatrix} \mathbf{T} \\ \mathbf{S} \end{pmatrix}$, where T is the time mapping, and S is the space (or processor) mapping. Thus, a particular iteration i will be executed by processor Si at time step Ti. Each element in the index set I requires certain data, typically expressed as subscripted array references. The time/space mapping of the index set imposes a time and space mapping on the generation and usage of the data elements. This synthesis process is also useful for mapping algorithms onto massively parallel computer systems.

Another approach for massive parallelism is to use an explicitly data parallel language model. The user is then responsible for mapping the data and execution to the processors (or virtual processors); some recent work on automating some of this process is encouraging [KLS90].

Here we focus on mapping an algorithm by applying program restructuring transformations, either automatically or semi-automatically (under user control). Restructuring transformations, such as loop interchanging, have been investigated as methods to discover additional parallelism in sequential programs [AlK84, Wol78, Wol82] and to optimize the performance of memory hierarchies [AKL81, IrT88]. We believe we can get many of the same benefits from program restructuring as from algebraic

mathods, without as many restrictions on the source program. This paper describes some restructuring transformations and how they would be used in such a scheme.

Model of Parallel Computation

We assume an ensemble of processors executing in either SIMD (Single Instruction Multiple Data) or SPMD (Single Program Multiple Data) mode. A parallel algorithm will consist of a number of steps, where each step comprises a computation phase and a communication phase. The computation phase describes parallel execution across all or a subset of the processor ensemble. An algorithm is expressed in the form of nested loops:

```
for i1 = ...
 for i2 = ...
   . . .
   for in = ...
    computation(i)
   endfor
   . . .
endfor
```

Two interpretations can be taken of such a program: an imperative program defines the order of execution of the iterations which is then used to find the data dependence relations, while a single-assignment program defines the data dependence relations which then are used to find a legal ordering of the iterations. The standard form of the *parallel* algorithm we use is:

```
do j1 = ...
 do j2 = ...
   . . .
   do j|T| = ...
    pardo k1 = ...
     pardo k2 = ...
       . . .
       pardo k|S| = ...
        computation(π(jk))
       endpardo
       . . .
    endpardo
    communication phase
   enddo
   . . .
enddo
```

where π is a mapping function from the **jk** space to the **i** space (π is $\mathbf{R}^{-1}$). The limits for each loop can in general be affine functions of the outer loop variables (and cannot be modified in the body of the loop); the loop variables will only be used in other loop limits and to index arrays or as literals within the body of the loop. In most of the examples, the communication phase will be implicit, but will be constrained to nearest neighbors according to the defined topology. While the original algorithm may be

either imperative or declarative, the target language is strictly imperative, meaning that the order of the execution of the iterations of the serial **do** loops is strictly forward.

More abstractly, the parallel algorithm can be viewed as:

```
do j1 = ...
 do j2 = ...
   . . .
   do j|T| = ...
    computation phase
    communication phase
   enddo
   . . .
enddo
```

where the computation phase is the set of nested **pardo** loops. The total number of loops in the parallel algorithm ($|T|+|S|$) must equal the number of loops in the original algorithm (n). The method described here will proceed by transforming the original algorithm by a series of elementary program transformations into an appropriate parallel form.

One difference between the approach described here and some other methods is that classical algebraic mapping methods only allow for a single time dimension, while our target algorithmic model allows nested sequential "time" loops. Formally, nested sequential loops implicitly define a single timing schedule via loop collapsing; to simplify the program restructuring approach, we keep the nested loops. Also, with nested loops some of the communication can be "floated" out of the innermost levels:

```
do j1 = ...
 do j2 = ...
   . . .
   do j|T| = ...
    computation phase
    communication phase
   enddo
   sub-communication phase
  enddo
  . . .
enddo
```

This still allows for regular communication patterns; taking advantage of local memory to reduce communication in this way can reduce the total load on the network bandwidth.

Note that we will take advantage of the topology of the processor network, much like other work in systolic array mapping strategies [CCL88, FoM85, LaM85, Qui84]. We envision our approach being applied to several models of processor ensembles, including SIMD parallel processors, systolic arrays of custom or semi-custom chips, or arrays of standard single-chip microprocessors.

Overview of Data Dependence

We say that nested loops define an *iteration space,* comprising a finite discrete Cartesian space with dimensionality equal to the loop nest level. For example, the loop below defines a two-dimensional 5X10 iteration space. In imperative languages, the semantics of a serial loop define the order in which the points in the iteration space are executed.

Program 1:

```
for i = 1 to 5
  for j = 1 to 10
    A(i,j) = B(i,j) + C(i)*D(j)
  endfor
endfor
```

There is no reason that the iteration space need be rectangular; many popular algorithms have inner loops whose limits depend on the values of outer loop indices. The iteration space for the loop below is triangular, suggesting the name triangular loop. Other interesting iteration space shapes can be defined by nested loops, such as trapezoids, rhomboids, and so on; we will show how some of these shapes can be generated from each other via loop restructuring.

Program 2:

```
for i = 1 to 5
  for j = i to 5
    A(i,j) = B(i,j) + C(i)*D(j)
  endfor
endfor
```

Data Dependence. Many compilers available today for vector and parallel computers advertise the ability to detect vector or parallel operations from sequential loops. Parallelism is detected by discovering the essential data flow (or data dependence) in the loop and allowing vector or parallel execution when data dependence relations are not violated. Loop restructuring transformations, such as loop interchanging, are often applied to enhance the available parallelism or otherwise optimize performance; data dependence information is needed to test whether restructuring transformations are legal (whether the program produces the same answer after restructuring as it did before).

In imperative languages, there are three essential kinds of data dependence. A *flow-dependence* relation occurs when the value assigned to a variable or array element in the execution of one *instance* of a statement is used (read, fetched) by the subsequent execution of an instance of the same or another statement. The loop below has a flow dependence relation from statement S_1 to itself, since the value assigned to A(i+1) will be used on the next iteration of the loop. We write this $S_1 \, \delta \, S_1$.

```
     for i = 1 to N-1
S1:    A(i+1) = A(i) + B(i)
     endfor
```

An *anti-dependence* relation occurs when the value read from a variable or array element in an instance of some statement is subsequently reassigned. In the loop below (assuming an imperative language), there is an anti-dependence relation from S_1 to S_2, since `B(i,j+1)` is used in S_1 and subsequently reassigned by S_2 in the next iteration of the `j` loop. We write this $S_1 \overline{\delta} S_2$.

Program 3:

```
     for i = 1 to N
       for j = 1 to M-1
S1:      A(i,j) = B(i,j+1) + 1
S2:      B(i,j) = C(i) - 1
       endfor
     endfor
```

Finally, an *output dependence* relation occurs when some variable or array element is assigned in an instance of a statement and subsequently reassigned. An example of this is shown below (again assuming an imperative language) where there is an potential output dependence relation from S_2 to S_1, since the variable `B(i+1)` assigned in S_2 may be reassigned in the next iteration of the loop by S_1. We write this $S_2 \delta^{o} S_1$. This also shows that the data dependence relations in a program must be approximated; since a compiler will not know the actual paths taken in the program, it must make conservative assumptions.

Program 4:

```
     for i = 1 to N-1
S1:    if(A(i) > 0) B(i) = C(i)/A(i)
S2:    B(i+1) = C(i) / 2
     endfor
```

Single Assignment Languages. In single-assignment languages, such as SISAL [MSA85] or Crystal [Che86], anti-dependence and output dependence relations cannot occur since variables and array elements cannot be reassigned. In Crystal, for instance, Program 4 would simply be illegal (since it tries to redefine `B(i)`), while Program 3 would be treated as having a flow dependence from S_2 to S_1, requiring the `j` loop to be executed backwards. For most of this paper we will concentrate on imperative language examples. Since our model of a parallel algorithm follows imperative semantics, programs written in an single-assignment language will be to converted to an imperative program. We will do this by loop transformations such that the normal imperative execution of the loops will satisfy all the dependence relations.

Distance and Direction Vectors. In order to apply a wide variety of loop transformations, data dependence relations are annotated with information showing how they are affected by the enclosing loops. Several such annotations are popular today. Many dependence relations have a constant distance in each dimension of the iteration space.

When this is the case, a *distance vector* can be built where each element is a constant integer representing the dependence distances in the corresponding loop. For example, in the following program there is a data dependence relation in the iteration space as shown in Figure 1; each iteration (i,j) depends on the value computed in iteration (i,j-1). We say that the distances for this dependence relation are zero in the i loop and one in the j loop, and we write this $S_1\ \delta_{(0,1)}\ S_1$.

Program 5:

```
     for i = 1 to N
       for j = 2 to M
S1:      A(i,j) = A(i,j-1) + B(i,j)
       endfor
     endfor
```

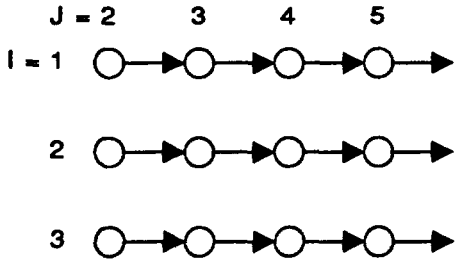


Figure 1.

For many transformations, the actual distance in each loop may not be so important as just the sign of the distance in each loop; also, the distance may not be constant, even though it may always be positive (or always negative). As an example, in the loop:

Program 6:

```
     for i = 1 to N
       for j = 1 to N
S1:      X(i+1,2*j) = X(i,j) + B(i)
       endfor
     endfor
```

the assignment to X(i+1,2*j) is used in some subsequent iteration of the i and j loops by the X(i,j) reference. Some of the dependence relations for this program are given in the table below:

	assigned by		used by		dependence
element	i	j	i	j	distance
X(2,2)	1	1	2	2	(1,1)
X(3,4)	2	2	3	4	(1,2)

The distance for this dependence in the j loop is always positive, but is not a constant. A common method to represent this is to save a vector of the signs of the dependence distances, called a *direction vector*. Each direction vector element will be one of

{+,0,-} [Ban88]; for historical reasons, these are usually written {<,=,>} [Wol78, WoB87, Wol89c]. In Program 3, the dependence relation would be written $S_1\ \delta_{(=,<)}\ S_1$.

Here we extend direction vectors by adding a fourth dependence direction explicitly for reductions. Take, for example, the imperative program:

```
     for i = 1 to N
S1:    S = S + A(i)
     endfor
```

Previous work in data dependence would classify this program as having the dependence relation $S_1\ \delta_{(<)}\ S_1$, thus implicitly preventing reordering of the index set. Using a single assignment language, this would have to be written:

```
     for i = 1 to N
S1:    S(i+1) = S(i) + A(i)
     endfor
```

which strictly defines the order of the accumulation. Because we want to be able to change the order of the accumulation, for associative reductions like this we will use the dependence relation $S_1\ \delta_{(R)}\ S_1$. This works equally well for certain declarative languages, such as Crystal, which do not define an order of accumulation in the first place. The difference is more noticeable with nested reductions:

```
     for i = 1 to N
       for j = 1 to M
S1:      S = S + B(i,j)
       endfor
     endfor
```

Previous work would say that this loop has the dependence relations $S_1\ \delta_{(<,*)}\ S_1$ as well as $S_1\ \delta_{(=,<)}\ S_1$. The dependence test for loop interchanging, for example, is that there must be no (<,>) directions; unfortunately, there is a (<,>) implied by the (<,*). With extended direction vectors, we call this $S_1\ \delta_{(R,R)}\ S_1$, a reduction in both dimensions. A reduction direction always corresponds to a dependence distance of one. Practically speaking, we will only be able to find reduction directions when all other directions are (=), and that is the only case we will deal with here.

Another popular data dependence annotation saves only the nest level of the outermost loop with a non-zero distance (non-(=) direction) [AlK87]. The dependence relation for Program 5 has a zero distance in the outer loop, but a non-zero distance in the inner loop, so we would write $S_1\ \delta^2\ S_1$. We also say that this dependence relation is *carried* by the inner j loop. Some dependence relations may not be carried by any loop, as below:

```
     for i = 1 to N
       for j = 2 to M
S1:      A(i,j) = B(i,j) + C(i,j)
S2:      D(i,j) = A(i,j) + 1
       endfor
     endfor
```

Here the references to A(i,j) produce a dependence relation from S_1 to S_2 with zero distance in both loops. We would thus say $S_1\ \delta_{(0,0)}\ S_2$ or $S_1\ \delta_{(=,=)}\ S_2$. Since it is carried by neither of the loops, we call it a *loop independent dependence,* represented $S_1\ \delta^{\infty}\ S_2$. This annotation by itself is too coarse for our application, though we will use the notion of loop-carried dependence in our discussion. In particular, a loop that does not carry any dependence relations can be executed in parallel; we will make use of this fact by transforming algorithms to change the loops which carry dependences, and moving dependence-carrying loops to the outermost levels.

For reduction loops, we say that the dependence relation is carried by all loops with an R direction. In the program:

```
     for i = 1 to N
       for j = 1 to M
         for k = 1 to L
S1:        T(j) = T(j) + B(i,j,k)
         endfor
       endfor
     endfor
```

the dependence relation is carried by the i and k loops, written $S_1\ \delta_{(R,=,R)}\ S_1$ or $S_1\ \delta^{1,3}\ S_1$.

Input Dependence. For our purposes there is reason to also look at the order in which data elements are read, when they are used multiple times. For instance, in Program 3, the array element C(i) on the right hand side of S_2 is used M-1 times, once for each iteration of the j loop. There is no data dependence constraint placed on the program by this fact; the different iterations can be assigned to different processors, and they can each fetch C(i) in any order; nonetheless, we will compute a relation between right hand side variable references, when any element is used more than once in the loop. For purposes of testing for legality of program restructuring as described in the next section, these *input dependence* relations will not disallow any transformation; however, we will keep these relations and update them as we update the other "real" data dependence relations. Whereas input dependence relations cannot have a reduction direction, we do note when they have correspond to a broadcast along some index dimension. So for Program 3 we say $S_2\ \delta^{I}_{(=,B)}\ S_2$. We will see in the next section that some transformations change (<) directions into (>) directions, which cannot be satisfied by imperative semantics (such as the target language), but input dependence relations will be handled in a special way.

Elementary Transformations

The element program restructuring transformations which we propose to use in our mapping method are

- loop interchanging,
- loop skewing,

- loop reversal and
- loop rotation.

For each transformation, we will describe its effect on data dependence relations and its effect on the shape of the iteration space. For the discussion below, assume that we have n nested loops, and that every dependence relation is annotated with a distance vector $(d_1, d_2, \ldots d_n)$ or direction vector $(\phi_1, \phi_2, \ldots, \phi_n)$. In all cases, d_k is either a constant integer, or is *, meaning it has unknown value, and $\phi_k \subset \{<, =, >, R\}$.

Loop Interchanging. Interchanging nested loops can dramatically change the execution characteristics of the loop, and can enhance the parallelism available at the inner loop levels [AlK84, Wol78, Wol82, Wol89c]. Loop interchanging essentially transposes an iteration space about the major diagonal; thus a 5X10 iteration space (Program 1) becomes a 10X5 space after interchanging, and an upper right triangular iteration space (Program 2) becomes a lower left triangle.

Loop interchanging can change the direction vector or distance vector associated with every data dependence relation in the loop. Loop interchanging of the k and k+1 loops changes the distance vector by interchanging the elements corresponding to those two loops, as in:

$$(d_1, d_2, \ldots, d_k, d_{k+1}, \ldots, d_n) \xrightarrow{\text{int}(k,k+1)} (d_1, d_2, \ldots, d_{k+1}, d_k, \ldots, d_n)$$

and the direction vector is modified as:

$$(\phi_1, \phi_2, \ldots, \phi_k, \phi_{k+1}, \ldots, \phi_n) \xrightarrow{\text{int}(k,k+1)} (\phi_1, \phi_2, \ldots, \phi_{k+1}, \phi_k, \ldots, \phi_n)$$

In a two-nested loop (n=2), for instance, interchanging the two loops would change a $(=, <)$ dependence relation into a $(<, =)$, moving the dependence-carrying loop outwards. However, a $(<, <)$ dependence relation would still look the same after interchanging, which would change the dependence carrying loop to the new outer loop. Notice that since we are allowing the order of reductions to change, reduction dependences do not prevent interchanging. Interchanging corresponds to the algebraic transformation $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

Any loop ordering is achievable by multiple applications of adjacent pairwise loop interchanging. There is often more than one way to change one loop ordering to another. For instance, a three-nested IJK loop can be changed to a KJI loop either by IJK→IKJ→KIJ→KJI or by IJK→JIK→JKI→KJI. We assume a practical algorithm for visiting all legal permutations of the loops without duplication (to reduce wasted time).

Loop Skewing. Loop skewing was introduced as a alternate derivation of the wavefront method [Wol86a], and we use it as such here. In the two-nested case below:

```
for i = 2 to N-1
 for j = 2 to N-1
  computation(i,j)
 endfor
endfor
```

the `j` loop can be skewed with respect to the `i` loop by adding `i` to the upper and lower limits of the `j` loop and subtracting `i` from `j` within the body of the loop:

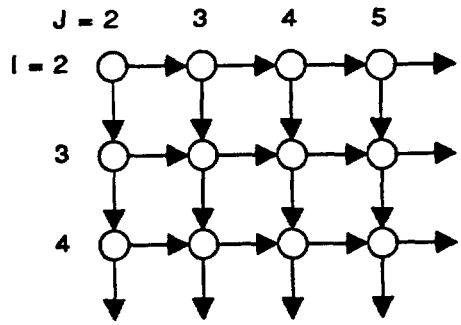
```
for i = 2 to N-1
 for j = i+2 to i+N-1
  computation(i,j-i)
 endfor
endfor
```

In general, a loop can be skewed with respect to any outer loop in which it is contained, and the loop can be skewed with any integer factor. For the examples in this paper, we will use only the skewing factors of `+1` or `-1` (the example above uses a factor of `+1`; a factor of `-1` would subtract `i` from the loop limits and add `i` within the body of the loop).

By itself loop skewing has no effect on the order of execution of the iterations of the loop. However it has significant impact on the data dependence relations in the loop. For instance, the following loop:

```
for i = 2 to N-1
 for j = 2 to N-1
  A(i,j) = A(i-1,j) + A(i,j-1)
 endfor
endfor
```

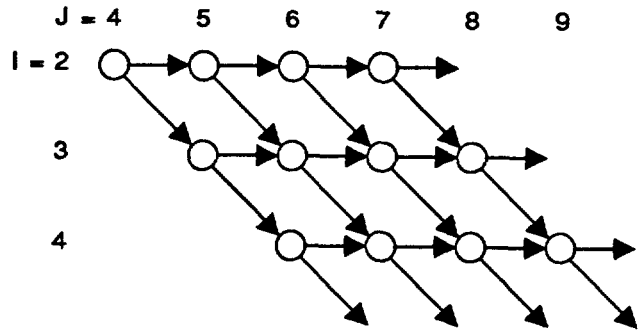
has two flow-dependence relations, with distance vectors `(1,0)` and `(0,1)` (or direction vectors `(<,=)` and `(=,<)`). The dependence relations in the iteration space are:



Since each loop carries a dependence relation, neither loop can be executed in parallel. However, skewing the inner loop by a factor of `+1` changes the program to:

```
for i = 2 to N-1
 for j = i+2 to i+N-1
  A(i,j-i) = A(i-1,j-i) + A(i,j-i-1)
 endfor
endfor
```

The new iteration space is a trapezoid:



In particular, the dependence relations have distance vectors (1,1) and (0,1) (or direction vectors (<,<) and (=,<)). Now consider what happens when the skewed loops are interchanged [Wol86b]:

```
for j = 2+2 to N-1+N-1
 for i = max(2,j-N+1) to min(N-1,j-2)
  A(i,j-i) = A(i-1,j-i) + A(i,j-i-1)
 endfor
endfor
```

Now the dependence distance vectors are (1,1) and (1,0) (direction vectors are (<,<) and (<,=)), meaning that the outer j loop carries both dependence relations. This means that the inner loop can be executed in parallel, since it no longer carries any dependence relations. This is the main effect of which we will take advantage by loop skewing. Another effect is simply aligning iterations differently in the processors, which will be useful to change interprocessor communication requirements, and prevent multiple processors from needing the same data at the same time.

As with loop interchanging, loop skewing can change the direction vector or distance vector associated with each dependence relation in the loop. Skewing the m loop with respect to the k loop by a factor of f will change the distance vector by adding the value of d_k to d_m:

$$(d_1, d_2, \ldots, d_k, \ldots, d_m, \ldots, d_n) \xrightarrow{\text{skew}(k,f,m)} (d_1, d_2, \ldots, d_k, \ldots, d_m + f d_k, \ldots, d_n)$$

In this paper, we will use only factors of ± 1. Skewing by a factor of +1 corresponds to the algebraic transformation $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$.

If the only information kept is a direction vector, skewing the m loop with respect to the k loop by a positive factor will change ϕ_m according to the following table:

positive skew		ϕ_m <	=	>	R	B
	<	<	<	*		
ϕ_k	=	<	=	>	R	B
	>	*	>	>		
	R		<		R	
	B		<			B

A reduction or broadcast entry acts like a < with a distance of one when combined with any other entry. If ϕ_k is (R) or (B), it will change to (<) unless $\phi_m \in \{R, B\}$; otherwise, ϕ_k remains the same. The corresponding table for skewing by a negative factor is:

negative skew		ϕ_m <	=	>	R	B
	<	*	>	>		
ϕ_k	=	<	=	>	R	B
	>	<	<	*		
	R		>		R	
	B		>			B

The * entries are due to the uncertainty of the sign of the sum a positive and negative number when the magnitude of those number is unknown. Loop skewing can be used to find additional parallelism when it can change a zero or negative direction in the inner loop to a positive direction, thus allowing loop interchanging to make that the carrying loop.

Loop Reversal. Loop reversal is simply running a loop backwards. It has little applicability when compiling imperative languages, but it will allow compilation of single-assignment programs to imperative semantics. In particular, when an single-assignment program requires a loop to run backwards, as below:

```
for i = 1 to N-1
 A(i) = A(i+1) + 1
endfor
```

reversal can allow normal imperative semantics to execute the loop sequentially:

```
for i = N-1 to 1 by -1
 A(i) = A(i+1) + 1
endfor
```

Reversal consists of switching the lower and upper limits and negating the increment of the loop.

When converting single-assignment programs to imperative programs, the order of execution of the iterations (forward or reverse) is defined by whether the loop carries any dependences. An imperative program can only carry a dependence with a positive distance (or (<) direction), so an single-assignment program with a loop carried

negative distance (as above) would have to be reversed. Not all single-assignment loops can be converted to imperative programs by simple inspection of dependence distances and loop reversal, but that is a subject for another paper.

Reversing the k loop negates the distance vector element d_k:

$$(d_1, d_2, \ldots, d_k, \ldots, d_n) \xrightarrow{\text{reverse}} (d_1, d_2, \ldots, -d_k, \ldots, d_n)$$

or the direction vector as:

$$(\phi_1, \phi_2, \ldots, \phi_k, \ldots, \phi_n) \xrightarrow{\text{reverse}} (\phi_1, \phi_2, \ldots, -\phi_k, \ldots, \phi_n)$$

where $-\phi_k$ is defined by:

	ϕ_k				
	<	=	>	R	B
$-\phi_k$	>	=	<	R	B

Reversal corresponds to the algebraic transformation $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$.

Loop Rotation. Loop rotation is most easily defined when the loops are "normalized", that is have a lower limit of zero and increment of one; unnormalized loops can always be normalized by a simple program transformation if desired [AlK87, KKP81]. Rotation is much like skewing [Wol89a]; in the two-nested case below:

```
for i = 0 to N-1
 for j = 0 to M-1
  computation(i, j)
 endfor
endfor
```

rotating the j loop with respect to the i loop changes the computation to:

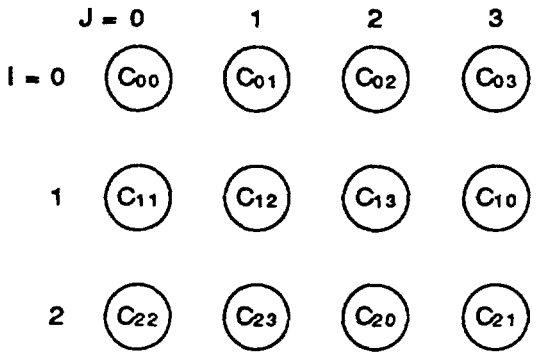
```
for i = 0 to N-1
 for j = 0 to M-1
  computation(i, (j-i) mod M)
 endfor
endfor
```

As with skewing, rotation by a factor of -1 is also legal, producing:

```
for i = 0 to N-1
 for j = 0 to M-1
  computation(i, (j+i) mod M)
 endfor
endfor
```

Loop rotation corresponds to skewing the loop around a torus; the picture below shows

a rotated iteration space where C_{ij} corresponds to *computation*(i, j), and N=3, M=4.



Loop rotation can also be applied to non-adjacent loops, as with loop skewing. Unlike skewing, rotation has no effect on the shape of the iteration space, since it merely moves part of the iteration space around.

Rotating loop m with respect to loop k will usually leave distance vector element d_k unchanged, since the order of execution of the iterations of the k loop are unchanged. However, two new distance vectors will be generated by rotation; for forward rotation (factor +1, where N is the trip count of the m loop), the two distance vector elements for the m loop will be $d_k+d_m \bmod N$ and $-(-d_k-d_m \bmod N)$.

$$(d_1,\ldots,d_k,\ldots,d_m,\ldots,d_n) \xrightarrow{\text{rotate(k,m)}} \begin{cases} (d_1,\ldots,d_k,\ldots,(d_k+d_m \bmod N),\ldots,d_n) \\ (d_1,\ldots,d_k,\ldots,-(-d_k-d_m \bmod N),\ldots,d_n) \end{cases}$$

The second distance vector holds for dependence relations that get rotated around the iteration space when d_k+d_m is not zero. We will use the notation d *rot* N to represent the two distances in the m^{th} position above. For direction vectors we can use the same table we had for skewing, except all < and > entries in the table must be replaced with *. If we do not know the dependence distances, then the direction vector elements for the interesting cases will be changed to:

(ϕ_k, ϕ_m) positive rotation		ϕ_m =	R	B	(ϕ_k, ϕ_m) negative rotation		ϕ_m =	R	B
	=	(=,=)	(=,R)	(=,B)		=	(=,=)	(=,R)	(=,B)
ϕ_k	R	(<,<>)	(R,R)		ϕ_k	R	(<,<>)	(R,R)	
	B	(<,<>)		(B,B)		B	(<,<>)		(B,B)

In other cases, any dependence carried by the m loop will be violated by loop rotation (in the sense that the imperative target language will violate that dependence). It is interesting to investigate what communication is implied when d_k is a small constant or ϕ_k is (R) or (B), and d_m is zero (ϕ_k is (=)). The k loop will still carry any

dependence (and thus be good for the time loop), and the communication implied by the dependence relations can be satisfied by end-around ring processor connections, using modular arithmetic, assuming the number of processors is the same as the number of iterations. In the table above, the < entries correspond to a dependence distance of one, and the <> entries correspond to a distance of (1 *rot* N).

It is also well defined to rotate a loop with respect to an inner loop. The simple two-nested computation:

```
for i = 0 to N-1
 for j = 0 to M-1
  computation(i, j)
 endfor
endfor
```

would be changed to:

```
for i = 0 to N-1
 for j = 0 to M-1
  computation( (i±j) mod M,j)
 endfor
endfor
```

After rotating outer loop k with respect to inner loop m, the dependence distance vector element d_k for will be $((d_k+d_m)\ rot\ M)$, where M is the trip count of the m loop. The direction vectors for the interesting cases will be:

(ϕ_k, ϕ_m) positive rotation		ϕ_m =	ϕ_m R	ϕ_m B	(ϕ_k, ϕ_m) negative rotation		ϕ_m =	ϕ_m R	ϕ_m B
	=	(=,=)	(<,<>)	(<,<>)		=	(=,=)	(<,<>)	(<,<>)
ϕ_k	R	(R,=)	(R,R)		ϕ_k	R	(R,=)	(R,R)	
	B	(B,=)		(B,B)		B	(B,=)		(B,B)

Again, the dependence distances corresponding to the < and <> entries are one with respect to rotation. Rotation, like skewing and interchanging, can change the loop which carries a dependence relation.

Restructuring Goals

We propose a method that will attempt to generate a comprehensive set of correct, equivalent forms of an input algorithm from which the best one or a set of "good" ones can be chosen. We start by looking at the goals or requirements of the final program.

Using the terminology in the first section, the time/space mapping $\begin{pmatrix} T \\ S \end{pmatrix}$ of the index set imposes a time and space mapping on the generation and usage of the data elements. If A is a data array, then index set element $i \in I$ refers to (or generates) $A(f_A(i))$, where f_A returns a vector of subscripts if A is multidimensional. Many algebraic mapping methods place restrictions on the form of f_A. Four goals that usually must be met by mapping methods are: The conditions that usually must be met

are:

1) No two iterations may be mapped to the same processor at the same time: for any two index set elements $\mathbf{i}, \mathbf{j} \in \mathbf{I}$ such that $\mathbf{i} \neq \mathbf{j}$, either $\mathbf{Ti} \neq \mathbf{Tj}$ or $\mathbf{Si} \neq \mathbf{Sj}$.
2) All data dependence relations must be satisfied: for any two index set elements $\mathbf{i}, \mathbf{j} \in \mathbf{I}$ such that $\mathbf{j}$ depends on $\mathbf{i}$, the relation $\mathbf{Ti} < \mathbf{Tj}$ must hold.
3) No two processors should require the same data at the same time: for any two index set elements $\mathbf{i}, \mathbf{j} \in \mathbf{I}$ such that $\mathbf{i} \neq \mathbf{j}$, if $f_A(\mathbf{i}) = f_A(\mathbf{j})$ (for any data array A), then $\mathbf{Ti} \neq \mathbf{Tj}$.
4) The index set must be mapped such that data will flow across the processor ensemble using available connections and the data motion across the systolic array must be uniform: for any data array A and any two index set elements $\mathbf{i}, \mathbf{j} \in \mathbf{I}$ such that $\mathbf{i} \neq \mathbf{j}$, $f_A(\mathbf{i}) = f_A(\mathbf{j})$ and $\mathbf{Ti} < \mathbf{Tj}$, then $\mathbf{Sj} = \mathbf{Si} + (\mathbf{Tj} - \mathbf{Ti})\, d_A \mathbf{v}_A$. The vector $\mathbf{v}_A$ is the direction of motion of the data array A, and must correspond to one of the available interprocessor connections; in a two dimensional mesh connected array, for instance, the four possible directions are $\begin{pmatrix}0\\1\end{pmatrix}$, $\begin{pmatrix}1\\0\end{pmatrix}$, $\begin{pmatrix}0\\-1\end{pmatrix}$ and $\begin{pmatrix}-1\\0\end{pmatrix}$. The value d_A is the delay of the array A, and often must be an integer greater than zero. Usually d_A will be one, meaning that the data moves from one processor to its neighbor every time step. Delays greater than one require extra registers or memory associated with each cell to hold the delayed values. Some recent work has proposed methods to allow data to "turn" under program control, or to allow multiple copies of read-only data to flow across the ensemble. Since our machine model includes a non-trivial amount of local memory, we only require that d_A be greater than or equal to one, but need not be integer; this can always be satisfied if the data communication is between neighbors.

Using program restructuring, the first goal (no two iterations can be mapped to the same point in space/time) is satisfied by the construction of the restructured program. Each transformation we use is one-to-one in the domain and range index sets. Moreover (as explained in Section 2), the mapping of the transformed program to time and space is as simple as mapping one or more dimensions of the transformed index set to time and the rest to space. Thus, no two iterations can possibly be mapped to the same space/time point.

The second goal (that all data dependence relations must be satisfied) can be met by requiring that no parallel loop can carry any dependence relation. The parallel loops in our computation model correspond to the space dimensions. Thus, the loops must be restructured and reordered so that the time loops carry all dependence relations; here we only worry about flow, anti and output dependence relations. It is a simple matter for a restructuring tool to check if the outer time loops carry all dependence relations, and to interchange or otherwise attempt to modify the program to guarantee this. Note that even if our input language is single-assignment, the output language is imperative (the sequential time loops run strictly forward, and the space "loops" run in asynchronously in parallel). Thus, the time loops can only satisfy loop carried dependence relations whose outermost non-zero dependence distance is positive (outermost non-(=) direction is (<) or (R)).

The third goal (no two processors need the same data at the same time) brings up the reason for computing and keeping input dependence relations. If two (or more) processors need a data element at the same time (as in Program 3), then there will be a loop-carried input-dependence relation. We can satisfy this goal by requiring that the time loops carry all input dependence relations also. Unlike the other kinds of dependence, however, we allow input dependence relations to have negative dependence distances or (>) directions in the carried loop position.

The fourth goal is much more difficult to satisfy. For instance, in systolic array synthesis, in order to guarantee uniform data flow across the processors, a restructuring tool must know the dependence distance in the space dimensions (to know the distance between the source and destination processors) and in the time dimensions (to compute the delay). For a more general computation, we may need only to know the distance in the space dimensions, if we know that distance is equal to one.

Global Reductions and Broadcasts. There may be applications where it is acceptable or desirable to allow the parallel loop (space loop) to compute a reduction. Some processor ensembles (such as hypercubes) have fast ways for all processors (or a subset of the processors) to compute a global reduction. If so, then we can relax rule 2 to allow the space dimensions to carry reduction dependence relations. Equally common are machines where the parallel loop is allowed to broadcast values along some axis or to all processors. In those cases, we can relax rule 3 to allow space dimensions to carry input dependence relations.

Two Examples

This section contains two simple examples to show the additional power of the program restructuring method compared to other automatic time/space mapping methods. The first example tries to map the Program 6 (from section 3) onto a general distributed memory processor ring. In addition, let all the data reside in the processors according to some mapping strategy to be computed automatically. The dependence relations are given in the table:

var	type	i	j
X	flow	1	<
B	input	=	B

where non-zero dependence distances are shown, if known. Because the target architecture is a ring (one dimensional), we will assign one loop to the space dimension and one to the time dimension. The time loop must satisfy (carry) all dependence relations, and the dependence distance in the space dimension must be known. This is satisfied if the loops are interchanged with the `j` loop corresponding to time and the `i` loop corresponding to space:

```
do j = 1 to N
  pardo i = 1 to N
    X(i+1,2*j) = X(i,j) + B(i)
  endpardo
enddo
```

In the mapped algorithm, each processor holds a column of X and an element of B. Condition 4 is satisfied by virtue of the distance in the space dimension being exactly one. At each time step j, each processor can send the j^{th} element of its column to its right neighbor, receive an element from its left neighbor, and compute element 2j of its column. Note that we used the same mapping strategy even though the target architecture was not systolic.

An interesting example is the automatic mapping of matrix multiply with perfect efficiency onto a two dimensional torus of processors. The original program is:

```
for i = 1 to N
  for j = 1 to N
    for k = 1 to N
      C(i,j) = C(i,j) + A(i,k)*B(k,j)
    endfor
  endfor
endfor
```

The goal is to have one time loop and two space loops (corresponding to the two dimensional processor topology) and perfect processor efficiency with no broadcasts or global reductions. The dependence table is:

	dependence	direction		
var	type	i	j	k
C	flow	=	=	R
A	input	=	B	=
B	input	B	=	=

No single loop carries all the dependence relations. However, the k loop carries the reduction; if we first *interchange* that loop to the outermost level, then *rotate* it with respect to the two inner loops, it will carry the input dependence relations also:

```
for k = 1 to N
  for i = 1 to N
    for j = 1 to N
      krot = (k+j-2) mod N + 1
      krot = (krot+i-2) mod N + 1
      C(i,j) = C(i,j) + A(i,krot)*B(krot,j)
    endfor
  endfor
endfor
```

The new dependence table is:

var	type	k	i	j
C	flow	R	=	=
A	input	1	=	1 *rot* N
B	input	1	1 *rot* N	=

The k loop can now serve as the time loop, so we have the desired form: the time loop carries all dependences and the dependence distance in the space loop is constant. This is generally known as "Cannon's form" of the matrix multiply, and is used on systems such as the Thinking Machines CM-2 [Joh87]. The advantage of having an automatic mechanism to discover this form of the algorithm is that the mechanism can then be applied to other loops and programs with similar dependence structure.

Restructuring Method

We propose a restructuring tool which implements the program restructuring transformations we have described based on the data dependence formalisms we have shown. The transformations are driven by a recursive control program which will exhaustively visit all the different possible restructured forms of the given algorithm. Each form will then be checked to see if it meets the minimum requirements for a legal space/time mapping, and some sort of optimality measure will be applied to choose the best mappings. We examine the problems of exhaustive search and explore possibilities to reduce the combinatorial explosion. The input to the tool is:

1. the algorithm (in either an single-assignment or imperative language);
2. limits on the target architecture, such as minimum and/or maximum dimensionality, size, and connectivity;
3. description of a measure of optimality, usually minimum time, but perhaps minimum space-time product or some other measure.

The control program will have the form:

```
function control( given : algorithm );
var new, best : algorithm;
var t : transformation;
  best := nil;
  if legal_mapping( given ) then
    best := given;
  endif;
  for t in transformation loop
    if legal( t, given ) then
      new := apply( t, given );
      new := control( new );
      if legal_mapping( new ) then
        if better( new, best ) then
          best := new;
        endif;
      endif;
    endif;
  endloop;
  return best;
end control;
```

Thus, at each level in the recursion, the control program will perform comparative analysis of all the legal mappings, and save the best. In practice a user might like to see several equivalently good mappings, or the top several mappings. Another flaw in this program is that the number of possible transformations at any one time depends on the form of the program; there are more possible loops to interchange with 4 nested loops than with only 2, for instance. Additionally, when the program recurses down a level it must ensure that it doesn't choose a transformation to nullify some previous transformation. Moreover, as was mentioned with respect to loop interchanging, there is often more than one sequence of loop transformations that can change an algorithm from form A to form B; the control program must prevent such redundant recursion.

Still, the size of the search tree is large. Let us explore this with a simple example. The following algorithm computes the value of a polynomial at many points X_i using the coefficients A_j by Horner's rule:

```
      for i = 1 to N
        for j = 1 to M
S1:       R(i) = R(i)*X(i) + A(j)
        endfor
      endfor
```

For this example, the target architecture is a linear array without global reduction or broadcast capability. The data dependence relations for this program are:

$$S_1 \ \delta_{(=,R)} \ S_1 \qquad \text{due to R}$$

$$S_1 \ \delta^I_{(=,<)} \ S_1 \qquad \text{due to X}$$

$$S_1 \ \delta^I_{(<,=)} \ S_1 \qquad \text{due to A}$$

The table of dependence relations is:

var	type	i	j
R	flow	=	R
X	input	=	1
A	input	1	=

We call this initial form of the algorithm the `IJ` form; Figure 2 shows part of the search tree of the different forms generated by automatic restructuring, where a superscript `R` means the loop was reversed, a subscript `+i` (`-i`) means the loop was skewed with respect to loop `i` by a factor of `+1` (`-1`), and a subscript of `r+i` or `r-i` means the loop was rotated with respect to `i`. At least 72 distinct forms of the algorithm can be generated by automatic means, many of which are legitimate mappings.

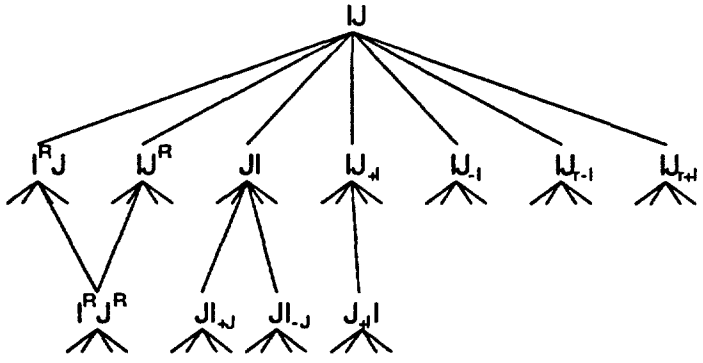


Figure 2.

Directed Restructuring. Exhaustive search of all possible restructurings quickly becomes too expensive. A method to prune the search tree is necessary. Here we explore in what cases each transformation is useful. Loop interchanging is useful for two reasons; first, it is obviously useful to bring a dependence-carrying loop to the outermost or "time" position. If some subset of the loops can be shown to carry all the dependence relations in the loop, then interchanging these to be the time loops will produce a legitimate mapping. In our example, no single loop carries all the dependence relations, so loop interchanging alone will not suffice. A second use for loop interchanging is to change the order of loops in preparation for skewing or rotation.

As we showed earlier, loop skewing can change an (=) dependence direction to a (<), and combined with interchanging can change the dependence-carrying loop and move it to the outer (time) position. In our Horner's rule program, for instance, skew-

ing the j loop with respect to i changes the dependence graph to:

var	type	i	j_{+i}
R	flow	=	R
X	input	=	1
A	input	1	1

and loop interchanging then produces:

var	type	j_{+i}	i
R	flow	R	=
X	input	1	=
A	input	1	1

Now, the outer loop carries all the dependence relations and can be used for the "time" loop, while the inner loop can be mapped to the "space" dimension:

```
      for j = 2 to N+M
        for i = max(1,j-M) to min(N,j-1)
S1:       R(i) = R(i)*X(i) + A(j-i)
        endfor
      endfor
```

Figure 3 shows a picture of how the computation proceeds through the first few time steps. The j loop could also be skewed with a factor of -1, which after interchanging would produce the dependence graph:

var	type	j_{+i}	i
R	flow	R	=
X	input	1	=
A	input	-1	1

Note that the elements of A are still buffered (remember that leading (>) directions are allowed for input dependence), but they will flow backwards:

```
      for j = 1-N to M-1
        for i = max(1,1-j) to min(N,M-j)
S1:       R(i) = R(i)*X(i) + A(j+i)
        endfor
      endfor
```

Figure 4 shows a picture of how this computation proceeds (for N,M=3).

Loop reversal is occasionally useful to enable loop interchanging by changing a (<,>) direction to a (<,<) direction. More practically, some applications (in a systolic architecture) may have constraints on the order in which data is presented to the processor ensemble. Here, loop reversal may be useful to invert the order in which the data elements are used (for instance, reversing the j loop in the j_{+i}i form above).

Loop rotation is another animal altogether; like skewing, rotation is useful for buffering dependence relations. The difference between rotation and skewing is that after loop rotation, when the outer loop is assigned to time, all the processors can begin

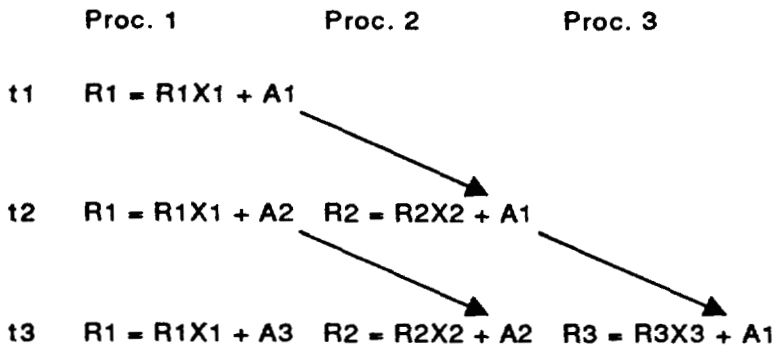


Figure 3.

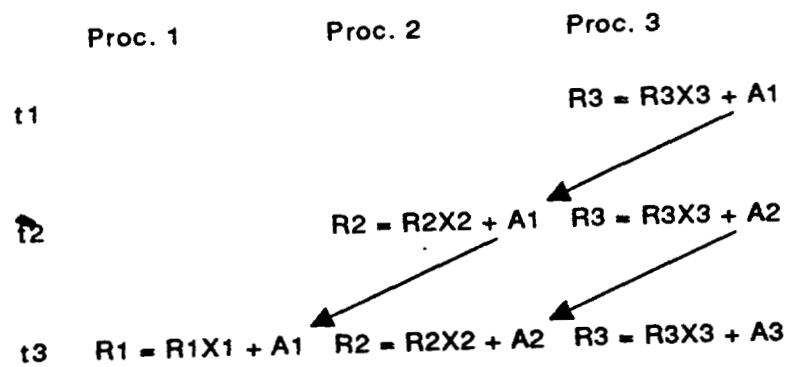


Figure 4.

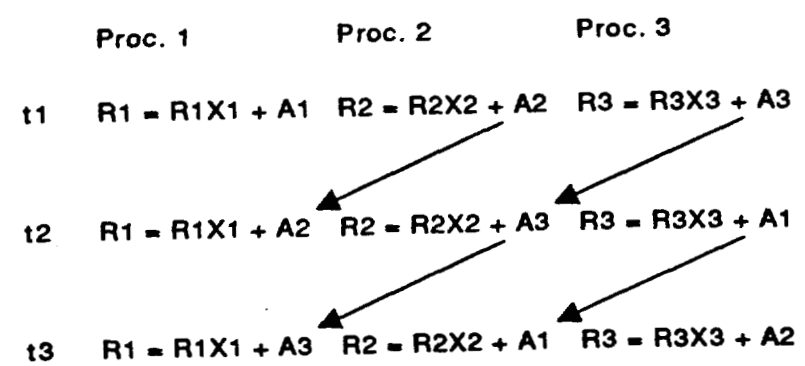


Figure 5.

simultaneously. There is no ramp-up and ramp-down at the beginning and end of the loop. This construct is useful in non-systolic applications, where the data already resides in the processors. Interchanging the j loop outwards and rotating it in the Horner's rule program above gives us (assuming M≥N):

```
        for j = 1 to M
          for i = 1 to N
            Jrot = (j+i-2) mod M + 1
S1:         R(i) = R(i)*X(i) + A(Jrot)
          endfor
        endfor
```

Figure 5 shows a picture of how this loop executes, after assigning it to the time dimension.

This analysis leads us to believe that the control program for the restructuring procedure can selectively choose to generate only a subset of restructured programs that are likely to generate legitimate mappings.

Summary and Limitations

We have shown a new approach to the problem of mapping a nested loop algorithm onto fixed topology processor ensembles. Our solution, based on program restructuring, is similar to other approaches based on finding linear program mappings from some representation of the data dependence relations in the program (often by algebraic manipulation). An important advantage to our approach is that it is not limited to linear mappings; although loop rotation is the only nonlinear transformation shown here, other transformations, such as folding or other contractions, can be defined and added to this approach [CCL88]. Another advantage is its flexibility; we are working on simple ways to use iteration space tiling to deal with fixed size processor ensembles or to improve the ratio between the computation inside a "time step" and the communication between steps [IrT88, Wol89b]. Finally, program restructuring is not limited to strict tightly-nested loop algorithms. By using advanced restructuring methods, our method will be enhanced to allow more general algorithmic forms, easing the programming task for a user [Wol86b].

One of the limitations of our approach is the restriction of skewing and rotating only by unit factors. The transformations are quite well-defined with larger magnitude factors, but the breadth of the search tree of restructured forms would grow uncontrollably if we tried many factors. We are looking at methods to directly compute the required skew or rotation factor from the program, e.g., from the dependence distances (when known). Without a direct method, we will be either confined to limit the skew factors allowed, or will have to potentially pay the cost of searching a large tree; this is directly equivalent to the method used to find the timing transformation π in [For88].

We make no claims that any such tool will ever be able to map "ordinary" programs into efficient code for a massively parallel system. We also claim that the "vectorization" approach will fail for such a system. A vectorizing compiler, when it finds a loop for which it cannot generate vector code, will generate scalar code; because the

ratio of vector to scalar performance on today's vector computers is in the range of 4-30, this is (more or less) acceptable for many users. For large parallel computers, the ratio of parallel to sequential performance is so great that generating "correct but slow" sequential code is as bad as or worse than failing altogether. We do claim that a user will be able to write a program in a convenient form with a familiar looking language, and with the aid of a tool will be able to run the program on massively parallel processor ensembles. We have begun work on a tool embodying our approach. Among the measures of success are the power, flexibility and efficiency (compile time) of the tool.

References

[AKL81] W. A. Abu-Sufah, D. J. Kuck and D. H. Lawrie, On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations, *IEEE Trans. on Computers C-30*, 5 (May 1981), 341-356.

[AlK84] J. R. Allen and K. Kennedy, Automatic Loop Interchange, in *Proc. of the SIGPLAN 84 Symposium on Compiler Construction*, New York, June 1984, 233-246.

[AlK87] J. R. Allen and K. Kennedy, Automatic Translation of Fortran Programs to Vector Form, *ACM Transactions on Programming Languages and Systems 9*, 4 (October 1987), 491-542.

[Ban88] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Norwell, MA, 1988.

[Che86] M. C. Chen, A Parallel Language and Its Compilation to Multiprocessor Machines or VLSI, in *Conf. Record of the Thirteenth Annual ACM Symp. on Principles of Programming Languages*, ACM Press, 1986, 131-139.

[CCL88] M. Chen, Y. Choo and J. Li, Compiling Parallel Programs by Optimizing Performance, *J. Supercomputing 2*, 2 (October 1988), 171-207, Kluwer Academic Publishers.

[FoM85] J. A. B. Fortes and D. I. Moldovan, Parallelism Detection and Transformation Techniques Useful for VLSI Algorithms, in *Journal Parallel & Distributing Computing*, 1985.

[For88] J. A. B. Fortes, On the Expansion, Analysis and Mapping of Conventional Programs into Code for Bit Level Processor Arrays, in *Proc. of Frontiers 88: the 2nd Symp. on the Frontiers of Massively Parallel Computation*, IEEE Computer Society Press, October 10-12, 1988, 567-574.

[IrT88] F. Irigoin and R. Triolet, Supernode Partitioning, in *Conf. Record of the 15th Annual ACM Symp. on Principles of Programming Languages*, ACM Press, New York, 1988, 319-329.

[Joh87] S. L. Johnsson, Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures, *J. of Parallel and Distributed Computing 4*, 2 (April 1987), 133-172.

[KLS90] K. Knobe, J. D. Lukas and G. L. Steele Jr., Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines, *J. of Parallel and*

ratio of vector to scalar performance on today's vector computers is in the range of 4-30, this is (more or less) acceptable for many users. For large parallel computers, the ratio of parallel to sequential performance is so great that generating "correct but slow" sequential code is as bad as or worse than failing altogether. We do claim that a user will be able to write a program in a convenient form with a familiar looking language, and with the aid of a tool will be able to run the program on massively parallel processor ensembles. We have begun work on a tool embodying our approach. Among the measures of success are the power, flexibility and efficiency (compile time) of the tool.

References

[AKL81] W. A. Abu-Sufah, D. J. Kuck and D. H. Lawrie, On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations, *IEEE Trans. on Computers C-30*, 5 (May 1981), 341-356.

[AlK84] J. R. Allen and K. Kennedy, Automatic Loop Interchange, in *Proc. of the SIGPLAN 84 Symposium on Compiler Construction*, New York, June 1984, 233-246.

[AlK87] J. R. Allen and K. Kennedy, Automatic Translation of Fortran Programs to Vector Form, *ACM Transactions on Programming Languages and Systems 9*, 4 (October 1987), 491-542.

[Ban88] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Norwell, MA, 1988.

[Che86] M. C. Chen, A Parallel Language and Its Compilation to Multiprocessor Machines or VLSI, in *Conf. Record of the Thirteenth Annual ACM Symp. on Principles of Programming Languages*, ACM Press, 1986, 131-139.

[CCL88] M. Chen, Y. Choo and J. Li, Compiling Parallel Programs by Optimizing Performance, *J. Supercomputing 2*, 2 (October 1988), 171-207, Kluwer Academic Publishers.

[FoM85] J. A. B. Fortes and D. I. Moldovan, Parallelism Detection and Transformation Techniques Useful for VLSI Algorithms, in *Journal Parallel & Distributing Computing*, 1985.

[For88] J. A. B. Fortes, On the Expansion, Analysis and Mapping of Conventional Programs into Code for Bit Level Processor Arrays, in *Proc. of Frontiers 88: the 2nd Symp. on the Frontiers of Massively Parallel Computation*, IEEE Computer Society Press, October 10-12, 1988, 567-574.

[IrT88] F. Irigoin and R. Triolet, Supernode Partitioning, in *Conf. Record of the 15th Annual ACM Symp. on Principles of Programming Languages*, ACM Press, New York, 1988, 319-329.

[Joh87] S. L. Johnsson, Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures, *J. of Parallel and Distributed Computing 4*, 2 (April 1987), 133-172.

[KLS90] K. Knobe, J. D. Lukas and G. L. Steele Jr., Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines, *J. of Parallel and*