Identifying Reordering Transformations That Minimize Idle Processor Time

Wayne Kelly and William Pugh
Department of Computer Science
University of Maryland, College Park, MD 20742
{wak,pugh}@cs.umd.edu

1 Introduction

The task of translating a sequential scientific program into an program that can be run efficiently on a parallel system involves making two key decisions. The first is how to distribute the iterations across the available processors. The second is the order in which the iterations should be executed. The primary goal when making the first decision should be to minimize interprocessor communication and to achieve good load-balance. A primary goal when making the second decision should be to minimize the idle time processors spend waiting for messages for other processors. These decisions cannot be made independently since an iteration ordering that may be good for one distribution of iterations to processors, may not be good for some other distribution.

We believe that the second of these decisions hasn't been adequately addressed by previous work in this area. We show that choosing a good ordering for the iterations can be extremely importantly and that this choice is heavy dependent on the way iterations are distributed. We show that existing approaches to this problem can produce results that are far from optimal. We will also describe analysis techniques that allow us to predict how good iteration orderings will be with respect to particular distributions of iterations.

We assume a SPMD model, where physical processors synchronize with other processors only when they need to communicate data. This synchronization can be performed by messages or by synchronization such as post and wait. Our compilation model is that iteration reordering transformations are applied to the original program and then the resulting program is transformed into SPMD code by specializing it so that only the iterations belonging to a particular processor are executed on it and the appropriate message handling and/or synchronizations is performed for interprocessor dependences.

The actual execution order of the iterations can make an enormous difference in the performance of the parallel program. Consider a stencil pattern computation with dependences up and to the right. If the array is decomposed into blocks of columns, and the outer loop iterates over columns, horrible performance will result: the second processor will not be able to start execution until the first processor is nearly completed. If the array is decomposed onto columns in a cyclic fashion, and the outer loop iterations over rows, horrible performance will result: a message/synchronization will occur between each stencil element computation.

This issue has been addressed in Fortran-D [7] by algorithms to identify "cross-processor loops" and a strategy of moving cross-processor loops inward for block distributions and outward for cyclic distributions. However, the definition of cross-processor loops is not theoretically sound. It works well for the stencil computations that typically arise in practice. But in some linear algebra kernels, it identifies all the loops as cross-processor loops, and hence provides no guidance. The strategy of how to move loops is a heuristic that frequently works well, but is not justified and may not be the only way to reorder loops so as to reduce idle time.

Some recent work [11] has suggested that block-cyclic distributions are important for obtaining good performance on some programs. In order for a iteration order to work well for a block-cyclic decomposition, it must (to a first approximation) also work well for both a cyclic and for a block decomposition. Thus the Fortran-D system offers no guidance in reordering computation for a block-cyclic decomposition. Since finding good execution orders for block-cyclic decompositions is more difficult, the more powerful and precise methods described in this paper are important.

A great deal of research has been done on deriving good layouts of data and/or computation [5, 1, 4]. In other systems such as Fortran-D and HPF [7, 6], the data decompositions are specified by the user and computation is decomposed according to an owner computes rule.

In this paper, we assume that we have been given a data and computation decomposition, and an ordering of the iterations of the program. We use this information to predict whether or not processors will idle waiting for synchronization from other processors. For each statement, we specify an affine mapping (referred to as a *space mapping*) from that statement's iteration space to a virtual processor space. This virtual processor space is then mapped to the physical processor space in either a blocked, cyclic or blocked-cyclic fashion. In this paper, we will only consider the case of one dimensional virtual and physical processor spaces. Figure 1 contains an example of a space mapping.

We specify an iteration re-ordering by specifying for each statement, a one-to-one mapping (referred to as a *time mapping*) from that statement's original iteration space to a new iteration space. The general form of a time mapping is:

$$T_p : [i_p^1, \ldots, i_p^{d_p}] \rightarrow [t_p^1, \ldots, t_p^n]$$

where

- The $i_p^1, \ldots, i_p^{d_p}$ are the index variables of the loops surrounding statement p.
- The t_p^k's (called mapping components) are affine functions of the index variables.

The time mappings dictate that if iteration i of statement p is mapped to the same physical processor as iteration j of statement q and $T_p(i)$ is lexicographically less than $T_q(j)$ then iteration i will be executed earlier than iteration j. This formalism is capable of representing any sequence of reordering transformations including (imperfectly nested) loop interchange, loop distribution, loop fusion and statement reordering [9].

Time mappings are not responsible for ensuring that data dependences between iterations on different physical processors are respected. These dependences must be enforced by inserting inter-processor synchronization at appropriate places. This synchronization can be implemented as either sends and receives if the address space is distributed, or posts and waits if the address space is shared. Time mappings must respect all data dependences. If intraprocessor dependences are violated by the time mappings, the wrong computations will be performed. If interprocessor dependences are violated by the time mappings, we cannot verify that the program is deadlock free.

We do not directly address the issue of how to derive a good reordering transformation that will minimize processor idle time, given a specific space mapping. However, since our evaluation can examine a prefix of the time mappings pronounce them as good, bad, or still-to-be-determined, we can use the methods described in [8] to derive a good reordering transformation.

Ideally, we would prefer a unified approach that would consider time and space mappings simultaneously, and derived an optimal combination. However, our understanding of the interaction between these two is inadequate for the loosely synchronized SPMD model where computation and communication can overlap. Once this interaction is better understood (which we hope to advanced in this paper), we may be able to derive an unified solution.

Original Program:

```
for i = 0 to n
    for j = 0 to m
        a(i,j) += a(i,j-1) + a(i-1,j)
```

Space mapping:

$S_1 : \{ [i,j] \rightarrow [i] \}$(blocked)

Figure 1: Example program and space mapping

2 Examples

Consider first the trivial program and space mapping shown in Figure 1. Table 1 shows the processor and time at which each iteration will be executed assuming: n = 3, m = 5, each iteration takes 1 time unit to execute and it takes 2 time units for a synchronization message to pass between processors. Results are shown for two different time mappings. Consider the first of these time mappings. Iteration $(0,2)$ is mapped to processor 1, but it depends on iteration $(0,1)$ which is mapped to processor 0. Processor 0 doesn't execute iteration $(0,1)$ until the 3^{nd} time step so iteration $(0,2)$ can't execute until time step 3 + latency = 5. In the mean time processor 1 waits idlely.

These results clearly show that the amount of time processors spend idle depends on the time mapping. Both of these time mappings result, to varying degrees, in pipelined execution - once each processor starts executing, it remains active until it has completed all of its iterations. In more complex examples, this is not always the case - a processor might execute some of its iterations and then have to wait for some other processor before it can execute more of its iterations.

We have developed a simulator that allows us to

	$\{ [i,j] \to [j,i] \}$			$\{ [i,j] \to [i,j] \}$		
Time	Proc0	Proc1	Proc2	Proc0	Proc1	Proc2
1	(0,0)			(0,0)		
2	(1,0)			(0,1)		
3	(0,1)			(0,2)		
4	(1,1)			(0,3)		
5	(0,2)	(2,0)		(1,0)		
6	(1,2)	(3,0)		(1,1)		
7	(0,3)	(2,1)		(1,2)		
8	(1,3)	(3,1)		(1,3)	(2,0)	
9		(2,2)	(4,0)		(2,1)	
10		(3,2)	(5,0)		(2,2)	
11		(2,3)	(4,1)		(2,3)	
12		(3,3)	(5,1)		(3,0)	
13			(4,2)		(3,1)	
14			(5,2)		(3,2)	
15			(4,3)		(3,3)	(4,0)
16			(5,3)			(4,1)
17						(4,2)
18						(4,3)
19						(5,0)
20						(5,1)
21						(5,2)
22						(5,3)

(n = 5, m = 3, latency = 2)

Table 1: Simulation of different time mappings

quickly obtain results similar to those in Table 1 for arbitrary programs. The most important piece of information in these results is the completion time (16 and 22 respectively for the examples shown in Table 1). Throughout this paper, we will give results of this form from our simulator. To isolate the effects of processor idling, we have ignored issues such as cache lines and network contention.

Consider again the program in Figure 1, this time with n = 255, m = 127 and 16 physical processors. Table 2 shows the results for various time mappings, methods of mapping virtual processors to physical processors and latencies. From these examples we can see that the method of mapping virtual processors to physical processors must be taken into account when deciding which time mapping to use. One could alternatively conclude that the time mapping must be taken into account when deciding how to map virtual processors to physical processors. However, the method of mapping virtual processors to physical processors is often dictated by other factors such as load balancing and minimizing communication. So, we prefer to take the view that we are given a method for mapping virtual processors to physical processors, and try to find the time mapping that best fits the circumstances. The dependences and time mappings in this example are symmetric, i.e. choosing a time mapping of $\{ [i,j] \to [i,j] \}$ given a space mapping of $\{ [i,j] \to [j] \}$ is analogous to choosing a time mapping of $\{ [i,j] \to [j,i] \}$ given a space mapping of $\{ [i,j] \to [i] \}$. So, this example also shows demonstrates that the space mapping must be taken into account when deciding which time mapping to use. We can also see from this example, that the importance of choosing a good time mapping, dramatically increases as the latency increases (a latency of 100 time units may in fact be optimistic for some distributed memory machines).

Time mapping: $\{ [i,j] \to [j,i] \}$

latency	block	cyclic(2)	cyclic
1	2288	28732	30863
3	2318	57660	92333
10	2407	156883	305077
100	3757	1438393	3049627

Time mapping: $\{ [i,j] \to [i,j] \}$

latency	block	cyclic(2)	cyclic
1	30622	16510	2047
3	30893	16893	2093
10	30998	17653	2677
100	32348	29212	25628

(n = 255, m = 127, procs = 16)

Table 2: Different virtual to physical mappings

Program 1:

```
for i = 0 to n
    for j = 0 to m
        a(i,j) += a(i-1,j-1) + a(i-1,j)
```

Program 2:

```
for i = 0 to n
    for j = i+0 to i+m
        a(i,j) += a(i-1,j-1) + a(i-1,j)
```

Figure 2: Different shaped iteration spaces

Now consider the two programs given in Figure 2. These programs have the same dependence pattern but different shaped iteration spaces. Table 3 shows simulation results for these two programs for two different time mappings. We see that neither time mapping is universally best, which demonstrates that the shape of the iteration space is another factor that needs to be considered when deciding which time mapping to use.

We have not yet explicitly demonstrated, but the most important factor that needs to be taken into account when deciding which time mapping to use is the dependence pattern. The dependences indicate which iterations must execute before which other iter-

Space mapping:

$S_1 : \{ [i,j] \rightarrow [i] \}$(blocked)

Time Mapping	Program 1	Program 2
$\{ [i,j] \rightarrow [j,i] \}$	2318	6158
$\{ [i,j] \rightarrow [i-j,i] \}$	6158	2318

(n = 255, m = 127, proc = 16, latency = 3)

Table 3: Different shaped iteration spaces

```
for k = 1 to n
  a(k,k) = sqrt(a(k,k))
    for i = k+1 to n
      a(i,k) = a(i,k) / a(k,k)
      for j = k+1 to i
        a(i,j) = a(i,j) - a(i,k)*a(j,k)
```

Figure 3: Cholesky Decomposition

ations and hence which processors may have to wait for which other processors. So, dependences not only dictate which time mappings are legal, they also affect which legal time mappings are good.

In the examples we have considered so far, the effects of each of the above mentioned factors can be easily factored out and analyzed separately. In real programs, these factors tend to combine together in complex ways. Real programs are more difficult to analyze than those we have seen so far because they usually contain multiple statements, each of which has its own space mapping and iteration space. Dependences can exist between different statements and can be non-uniform. Table 4 contains simulation results for the more realistic program shown in Figure 3. To simplify the table, we report the performance as overheads on top of the time required if the computations were evenly divided and no dependences existed. The imbalance overhead is the imbalance caused by physical processors having uneven workloads assigned to them; it reflects the execution time without dependences but with the work decomposition specified by the space mapping. The overheads for specific time/space mappings give the additional overhead when dependences are enforced. a d in the first column indicates that a loop was distributed even though it did not need to be distributed.

On first inspection, the good time mappings seem to contain no discernable characteristics to distinguish them from poor time mappings. In Section 4 we will describe analysis techniques that allow us to predict which of these time mappings are good.

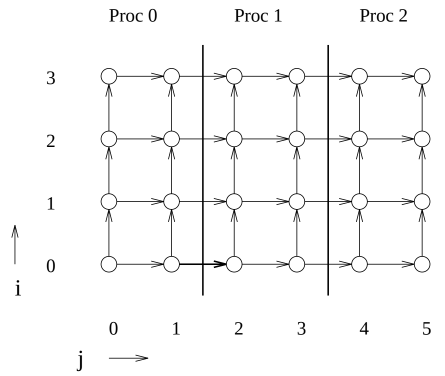

Figure 4: Iteration space and dependences

3 Virtual to Physical Mapping

We will assume that virtual processor 0 is the lowest numbered virtual processor used. We define the function *own* as:

$$v \in own(p) \text{ iff } p = \lfloor v/B \rfloor \bmod P$$

where P is the number of physical processors and B is a specified constant. This function defines the mapping from virtual to physical processors. If $v \in own(p)$, then virtual processor v is executed on physical processor p. $B = 1$ produces a cyclic distribution, $B = \lceil(\text{max virtual processor})/P\rceil$ produces a blocked distribution, and everything in between produces a block-cyclic distribution. In many cases, we will not actually know P or even B at compile time. Usually, the functions we derive are not dependent on the precise values of these variables (although they do depend considerably on whether $B = 1$ or $B > 1$).

4 Analyzing Time Mappings

Consider again the program in Figure 1. Figure 4 shows the iteration space and dependences of this program mapped onto 3 processors for the case of n = 5, m = 3. For the purposes of analyzing processor idle time, we need only analyze dependences that cross physical processor boundaries. Consider the dependence from iteration $(0,1)$ to iteration $(0,2)$. If we use a time mapping of $\{ [i,j] \rightarrow [i,j] \}$ then in the transformed program, iteration $(0,1)$ will be executed in the 1^{st} iteration of the outermost loop on processor 0 and iteration $(0,2)$ will be executed in the 1^{st} iteration of the outermost loop on processor 1. So, with respect to the outermost loop of the transformed program, processor 1 will not have to wait for processor 0. If we use a time mapping of $\{ [i,j] \rightarrow [j,i] \}$

	first dimension distributed: space mapping			cyclic			cyclic(2)		
	$S_1[k] \rightarrow [k]$	$S_2[k,i] \rightarrow [i]$	$S_3[k,i,j] \rightarrow [i]$	imbalance 4%			imbalance 7%		
	time mapping			latency					
	$T_1[k] \rightarrow$	$T_2[k,i] \rightarrow$	$T_3[k,i,j] \rightarrow$	1	32	64	1	32	64
KIJ	$[k,0,0,0,0]$	$[k,1,i,0,0]$	$[k,1,i,1,j]$	0%	4%	13%	0%	16%	59%
KIJd	$[k,0,0,0]$	$[k,1,i,0]$	$[k,2,i,j]$	0%	1%	4%	0%	1%	3%
KJId	$[k,0,0,0]$	$[k,1,i,0]$	$[k,2,j,i]$	0%	1%	3%	0%	0%	2%
IKJ	$[k,1,0,0,0]$	$[i,0,k,0,0]$	$[i,0,k,1,j]$	0%	0%	0%	96%	97%	98%
IJK	$[k,1,0,0,0]$	$[i,0,k,1,0]$	$[i,0,j,0,k]$	0%	0%	0%	100%	101%	102%
JKI	$[k,1,0,0]$	$[k,2,i,0]$	$[j,0,k,i]$	0%	2%	5%	0%	1%	2%
JIK	$[k,1,0,0]$	$[k,2,i,0]$	$[j,0,i,k]$	0%	2%	5%	0%	1%	2%
	second dimension distributed: space mapping			cyclic			cyclic(2)		
	$S_1[k] \rightarrow [k]$	$S_2[k,i] \rightarrow [k]$	$S_3[k,i,j] \rightarrow [j]$	imbalance 0%			imbalance 0%		
	time mapping			latency					
	$T_1[k] \rightarrow$	$T_2[k,i] \rightarrow$	$T_3[k,i,j] \rightarrow$	1	32	64	1	32	64
KIJ	$[k,0,0,0,0]$	$[k,1,i,0,0]$	$[k,1,i,1,j]$	4%	6%	8%	6%	7%	8%
KIJd	$[k,0,0,0]$	$[k,1,i,0]$	$[k,2,i,j]$	1%	4%	6%	6%	7%	8%
KJId	$[k,0,0,0]$	$[k,1,i,0]$	$[k,2,j,i]$	1%	4%	6%	6%	7%	8%
IKJ	$[k,1,0,0,0]$	$[i,0,k,1,0]$	$[i,0,j,0,k]$	4%	63%	193%	108%	173%	242%
IJK	$[k,1,0,0,0]$	$[i,0,k,0,0]$	$[i,0,k,1,j]$	10%	143%	285%	10%	75%	143%
JKI	$[k,1,0,0]$	$[k,2,i,0]$	$[j,0,k,i]$	0%	0%	0%	105%	106%	107%
JIK	$[k,1,0,0]$	$[k,2,i,0]$	$[j,0,i,k]$	282%	284%	287%	291%	292%	293%

Table 4: Cholesky Decomposition (n = 128, procs = 4)

then in the transformed program, iteration $(0,1)$ will be executed in the 2^{nd} iteration of the outermost loop on processor 0 and iteration $(0,2)$ will be executed in the 1^{st} iteration of the outermost loop on processor 1. So, processor 1 will have to wait for processor 0 to execute an entire iteration of its outermost loop. This explains why $\{ [i,j] \rightarrow [i,j] \}$ is a better time mapping in this situation than $\{ [i,j] \rightarrow [j,i] \}$. Some waiting occurs even if we use the $\{ [i,j] \rightarrow [i,j] \}$ time mapping. We saw that iterations $(0,1)$ and $(0,2)$ are executed in the same iteration of the outermost loop using this mapping, but iteration $(0,1)$ is executed in the 2^{nd} iteration of the innermost loop, while iteration $(0,2)$ is executed in the 1^{st} iteration of the inner most loop. So, processor 1 will have to wait for processor 0 to execute an iteration of its innermost loop. This is not ideal, but it is the best we can do in this situation.

The dependences we are given are specified with respect to the original global iteration space. To analyze processor idle time, we need to convert iterations in this global space into local iterations (also referred to as local time) on each processor. For example, if we are using the $\{ [i,j] \rightarrow [i,j] \}$ time mapping then global iteration $(0,1)$ becomes local iteration $(0,1)$ on processor 0 and global iteration $(0,2)$ becomes local iteration $(0,0)$ on processor 1. Similarly, if we are using the $\{ [i,j] \rightarrow [j,i] \}$ time mapping then global iteration $(0,1)$ becomes local iteration $(1,0)$ on processor 0 and global iteration $(0,2)$ becomes local iteration $(0,0)$ on processor 1. The idea of local time is that iterations on different processors will execute at the same "wall clock time" (assuming there is no waiting) if and only if the iterations have the same local time. What we are actually interested in is the difference in the local times of the iterations involved in inter-processor dependences. This will give us the amount of time one processor will have to wait for the other assuming there have been no delays so far. For example, the dependence from $(0,1)$ to $(0,2)$ has a local difference of $(0,-1)$ and $(-1,0)$ respectively, for the two time mappings considered. We use the lexicographic ordering of these local differences to rank time mappings (negative is bad, positive is good). We do this for all inter-processor dependences and then combine the results (as explained in Section 5 to obtain an overall assessment of the time mapping. The next section explains the general method we use to compute these local differences.

4.1 Local time and differences

To convert from a point in the original global iteration space to a point in a local iteration space, we first convert to a point in the transformed global iteration space by applying the appropriate time mapping. We then convert this point to a point in a local iteration space by considering; the shape of the iteration space, the space mapping and the method of mapping virtual processors to global processors. Figure 5 illustrates the logical steps we perform to convert a dependence in the original global iteration space, to a difference in

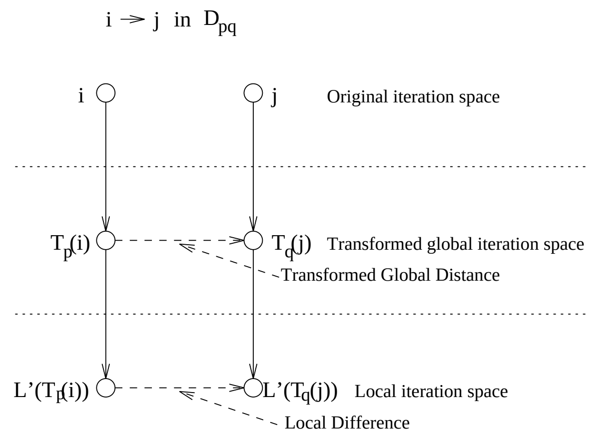

Figure 5: Global to local differences

local time.

We compute local times and local differences one level at a time, starting at the first/outermost level. That is, if our time mappings are of the form:

$$T_p : [i_p^1, \dots, i_p^{d_p}] \to [t_p^1, \dots, t_p^n]$$

then we first compute the 1^{st} of components of local times and differences using the 1^{st} components (t_p^1 of each statement p) of the time mappings, then we compute the 2^{nd} components of the local times and differences using the first 2 components (t_p^1 and t_p^2 of each statement p) of the time mappings, and so on, up to level n. The rest of this section describes how to compute k^{th} component of the local times and differences.

4.2 Groups of statements

It is only meaningful to compute local difference up to the common nesting level in the transformed program of the two statements. At each level we partition the statements into groups that indicate whether those statements will be executed in the same loop at that level. If all of the iterations of one statement are executed before all of the iterations of another statement at some level, then we assume that those statements will be executed by different loops at that level and hence will be in different groups; otherwise they are in the same group. At each level, all dependences that haven't been carried by earlier levels will be between statements within the same group. The rest of this section describes how to compute local times and differences for all dependences within a given group of statements G.

4.3 Local time for one level

We use the expression $L(t,p)$ to represent the k^{th} component of *local time* for all iterations i such that i is an iteration of some statement q, $t_q^k(i) = t$ and $S_q(i) = p$. We compute the set

$$\begin{aligned} K = \{\ t\ |\exists i, p \text{ s.t. } p \in G \wedge i \in I_p \wedge t = t_p^k(i) \wedge \\ S_p(i) = v \wedge t_p^1(i) = c_1 \wedge \dots t_p^{k-1}(i) = c_{k-1}\} \end{aligned}$$

representing range of values that will be executed by the level k loop for statements in G on virtual processor v ($c_1, \dots, c_{k-1}$ are symbolic constants representing the values of outer level loop variables).

We compute local time by starting with the global time and normalizing where necessary. If the lower bounds on t in K are independent of v then no normalization is necessary and t can be used as the local time for all statements in G. If the lower bounds on t in K do involve v then one of the following forms of normalization will be required. If there are multiple lower bounds on t in K then we need to apply the following normalization techniques to derive a result for each lower bound and use the most pessimistic result.

If the set K implies a constraint $t = \alpha v + \beta$, then each virtual processor executes a single iteration of the loop and the set of local iterations may be non-continuous. In this case we use:

$$L(t,p) = \left(\left\lfloor \frac{t-\beta}{\alpha B P} \right\rfloor, \frac{t-\beta}{\alpha} \bmod B \right)$$

as the local time. Note that L's result is a tuple, which should be interpreted lexicographically. As it turns out, L does not depend on p in this case.

If the set K contains more than one value, then, except for border cases, the range of values executed on each physical processor will be continuous. In this case we use:

$$L(t,p) = t - \min_{v \in own(p)} cv$$

as the local time, where c is the coefficient of v in the lower bound on t in K.

4.4 Local differences for one level

Given a function for $L(t,p)$, we can now compute local differences. We first define the function:

$$E(d,p,p') = \min_{t \in K} L(t+d, p') - L(t,p)$$

which gives the minimum local difference between any two iterations executed on physical processors p and p' respectively, that are separated by a global distance of d. We use this function to define:

$$\begin{aligned} F(d,s) = \min\{E(d,p,p')\ |\exists v \text{ s.t. } v \in own(p) \wedge \\ v + s \in own(p') \wedge p \neq p'\} \end{aligned}$$

which gives the minimum local difference between any two iterations executed on virtual processors s apart and separated by a global distance of d.

Finally, this function is used to define:

$$\begin{aligned}M(p,q,D) =&\min\{F(d,s) \mid \exists i \rightarrow j \in D \wedge \\ & t_q^k(j) - t_p^k(i) = d \wedge S_q(j) - S_p(i) = s\}\end{aligned}$$

the minimum value of the k^{th} component of the local difference for all dependences in dependence relation D (between statements p and q). Dependence Relations are the abstractions that we use to describe dependences. They consist of a set of affine constraints that describe exactly which iterations are dependent on which other iterations. For example, the dependences in Figure 4 would be represented by the dependence relation:

$$\begin{aligned}&\{[i,j] \rightarrow [i',j'] \mid \\ &\quad (i' = i \wedge j' = j+1 \wedge 0 \le i \le 5 \wedge 0 \le j \le 3) \vee \\ &\quad (i' = i+1 \wedge j' = j \wedge 0 \le i \le 5 \wedge 0 \le j \le 3)\}\end{aligned}$$

The function $M(p,q,D)$ is applied to all dependence relations D between statements in G that have interprocessor dependences (i.e., $i \rightarrow j$ s.t. $S_p(i) \neq S_q(j)$). The results are then combined as explained in Section 5 to obtain an overall assessment of the time mapping. We now consider how to compute $E(d,p,p')$ and $F(d,s)$ in various settings.

4.4.1 Non-continuous loops

If K contains only a single value then

$$\begin{aligned}F(d,s) = &\min_t \Big(\left\lfloor \tfrac{t+d-\beta}{\alpha BP} \right\rfloor - \left\lfloor \tfrac{t-\beta}{\alpha BP} \right\rfloor, \\ &(\tfrac{t+d-\beta}{\alpha} \bmod B) - (\tfrac{t-\beta}{\alpha} \bmod B)\Big) \\ = &\left(\left\lfloor \tfrac{d}{\alpha BP} \right\rfloor, 1 - B + (\tfrac{d}{\alpha} - 1 \bmod B)\right)\end{aligned}$$

If P, B or d are not known exactly, then we assume the worst case result of $(0, 1-B)$.

4.4.2 Continuous loops

If K contains more than a single value then

$$\begin{aligned}E(d,p,p') &= \min_t L(t+d,p') - L(t,p) \\ &= d + \min_{v \in own(p)} cv - \min_{v \in own(p')} cv\end{aligned}$$

If c is positive:

$$\begin{aligned}E(d,p,p') &= d + c(\min_{v \in own(p)} v - \min_{v \in own(p')} v) \\ &= d + c((Bp) - (Bp')) \\ &= d - cB(p'-p)\end{aligned}$$

If c is negative:

$$\begin{aligned}E(d,p,p') &= d + c(\max_{v \in own(p)} v - \max_{v \in own(p')} v) \\ &= d + c((Bp + B - 1) - (Bp' + B - 1)) \\ &= d - cB(p'-p)\end{aligned}$$

$$\begin{aligned}F(d,s) &= d - cBf(s) && \text{,if } c > 0 \\ &= d + cBf(-s) && \text{,if } c < 0\end{aligned}$$

where:

$$\begin{aligned}f(s) = \max\{p'-p \mid \quad & \exists v \text{ s.t. } v \in own(p) \wedge \\ & v + s \in own(p') \wedge p \neq p'\}\end{aligned}$$

If a pure block decomposition is used:

$$\begin{aligned}f(s) &= -1 && \text{,if } -B \le s \le 1 \\ &= \lceil \tfrac{s}{B} \rceil && \text{,otherwise}\end{aligned}$$

For cyclic or block cyclic decompositions, we approximate $f(s)$ when $s > BP - B$ because $f(s)$ is very sensitive to the exact values of B, P and s.

$$\begin{aligned}f(s) &= P - 1, \text{if } BP - B < s \\ &= \lceil \tfrac{s}{B} \rceil, \text{if } 0 < s \le BP - B \\ &= P + \min(-1, \lfloor (B-1+s)/B \rfloor), \text{if } s < 0\end{aligned}$$

4.5 Example revisited

We now apply these formal methods to the example in Figure 1 with a time mapping of $\{ [i,j] \rightarrow [i,j] \}$. The transformed iteration space has two levels. We first compute local differences at level 1:

$$G = \{ 1 \} \text{ and } K = \{ t \mid 0, v \le t \le v, 5 \}$$

The lower bound on t in K does depend on v so normalization is required. Since K contains only a single value we apply the non-continuous case:

$$F(d,s) = \left(\left\lfloor \frac{d}{6} \right\rfloor, 1 - 2 + (d - 1 \bmod 2) \right)$$

$$\begin{aligned}D = \{[i,j] \rightarrow [i',j'] \mid \quad & i' = i+1 \wedge j' = j \wedge \\ & 0 \le i \le 5 \wedge 0 \le j \le 3\}\end{aligned}$$

$$\begin{aligned}&M(1,1,D) \\ &= min\left\{ \left(\left\lfloor \tfrac{d}{6} \right\rfloor, -1 + (d-1 \bmod 2)\right) \mid \exists i,j,i',j' \text{ s.t.} \right. \\ &\quad i' = i+1 \wedge j' = j \wedge 0 \le i \le 5 \wedge 0 \le j \le 3 \wedge \\ &\quad \left. i' - i = d \wedge i' - i = s \right\} \\ &= (0, -1)\end{aligned}$$

We next compute local differences at level 2:

$$G = \{ 1 \} \text{ and } K = \{ t \mid 0 \le t \le 3 \}$$

The lower bound on t in K does not depend on v so no normalization is required.

$$F(d,s) = d$$

$$D = \{[i,j] \rightarrow [i',j'] | \quad i' = i+1 \wedge j' = j \wedge \\ 0 \leq i \leq 5 \wedge 0 \leq j \leq 3\}$$

$$M(1,1,D) \\ = min\{\ d \mid \exists i,j,i',j' \text{ s.t. } i' = i+1 \wedge j' = j \wedge \\ 0 \leq i \leq 5 \wedge 0 \leq j \leq 3 \wedge j' - j = d \wedge i' - i = s\} \\ = 0$$

Final result is local distance: $(0, -1, 0)$

5 Pipelining

If our analysis determines that processors will need to wait for data being computed on other processors, we need to consider how often such waiting occurs. For example, for the program in Figure 1 and Table 1, each successive processor incurs one more synchronization cost than the processor to its left. We describe this as pipelined computation. If our analysis in Section 4 determines that we will not need to wait on synchronization, it doesn't matter whether or not the computation is pipelined.

For stencil computations over a pure blocked decomposition, there is a simple rule for analyzing pipelining. If all interprocessor dependences are strictly increasing in physical processor number, then the delay for the entire program is $(P-1)d$, where d is the maximum delay for a single synchronization. If this situation occurs only once interprocessor dependences carried by outer loops have been removed, then the delay needs to be multiplied by the number of iterations of those outer loops.

This pipelining rule can rarely be applied to cyclic or block-cyclic decompositions, since even if the dependences are strictly increasing in virtual processor number, they wrap around in physical processor number (and are not strictly increasing in physical processor number). However, pipelining does occur in cyclic computations.

In stencil computations over cyclic decompositions (which is probably an unrealistic case), another behavior is seen. If a physical processor executes its virtual processors in order (e.g., processor p executes all the iterations belonging to virtual processor p, then the iterations belonging to virtual processor $p+P$ ...), pipelining can occur in a way very similar to pipelining for block stencil computations. In order for this to happen, the work done by each virtual processor must be greater that Pd, where d is the synchronization delay between virtual processors. If this is true, then virtual processor $p+P$ will be enabled by the time physical processor p completes virtual processor p's work.

We have also been developing techniques to analyze pipelining-like behavior observed in some loop orderings of linear algebra kernels such as Cholesky decomposition and Gaussian elimination with a cyclic or block-cyclic space mapping. However, at this point our experience is too slight and our data points too few to have sufficient confidence that these techniques will accurately predict the behavior of programs we haven't studied yet. We are continuing to develop these techniques and hope to present them soon.

6 Results

Table 5 contains the predictions made by our algorithm for Cholesky decomposition. We consider distribution of both the first and second dimension, and a cyclic and cyclic(2) folding function. The only case where we need to apply an adjustment due to lower bounds being dependent on v are in the IKJ and IJK loop orderings with the second subscript distributed. Since v is equal to j and $i \geq j$, we have $c = 1$ and $f(s) = P - 1$ for an offset of -3. The adjustment for non-continuous loops need to be applied in many of the examples.

Our predictions are fairly accurate; the only serious misprediction is that is that we predict that using a `a(:,cyclic)` distribution and the JKI loop ordering will be bad, when in fact it is quite good. The cases where we make no prediction turn out to be good, fair or bad. We have been able to predict all of these cases more accurately using our methods for analyzing pipelined behavior, mentioned in Section 5. Without taking pipelining into account, all loops predicted to be good are in fact good. Pipelining can only save potentially bad performance; it cannot hurt a good loop order.

7 Related Work

The only work to directly address this issue is the work on cross-processor loops in Fortran D [7]. Fortran D normally uses an owner computes rule and user-supplied data decompositions. An algorithm is given in [7] to identify "cross-processor loops". Cross-processor loops are informally defined as:

> Sequential space-bound loops causing computation wavefronts that cross processor boundaries (*i.e.*, sweeps over the distributed dimensions).

For a block decomposition, cross-processor loops are interchanged inward; for a cyclic decomposition cross-processor loops are interchanged outward. While the definition and strategy work for stencil computations, it is not theoretically sound and the conditions it

First subscript distributed

Synchronizing dependence : $\{[k,i] \rightarrow [k,i',i] : 1 \leq k < i < i' \leq n\}$

	global transformed		local transformed difference		prediction	
version	dependence	difference	cyclic	cyclic(2)	cyclic	cyclic(2)
KIJ	$[k,1,i,0] \rightarrow [k,1,i',1,i]$	[0,0,+,1]	[0,0,0+,1]	[0,0,0+,-1,1]	?	bad
KIJd	$[k,1,i] \rightarrow [k,2,i',i]$	[0,1]	[0,1]	[0,1]	good	good
KJId	$[k,1,i] \rightarrow [k,2,i,i']$	[0,1]	[0,1]	[0,1]	good	good
IKJ	$[i,0,k,0] \rightarrow [i',0,k,1,i]$	[+,0,0,1]	[0+,0,0,1]	[0+,-1,0,0,1]	?	bad
IJK	$[i,0,k,1] \rightarrow [i',0,i,0,k]$	[+,0,+,-1]	[0+,0,+,-1]	[0+,-1, 0,+,-1]	good	bad
JKI	$[k,2,i] \rightarrow [i,0,k,i']$	[+,-2]	[+,-2]	[+,-2]	good	good
JIK	$[k,2,i] \rightarrow [i',0,i,k]$	[+,-2]	[+,-2]	[+,-2]	good	good

Second subscript distributed

Synchronizing dependence: $\{[k,i] \rightarrow [k,i,j] : 1 \leq k < j \leq i \leq n\}$

	global transformed		local transformed difference		prediction	
version	dependence	difference	cyclic	cyclic(2)	cyclic	cyclic(2)
KIJ	$[k,1,i,0] \rightarrow [k,1,i,1,j]$	[0,0,0,1]	[0,0,0,1]	[0,0,0,1]	?	?
KIJd	$[k,1,i,0] \rightarrow [k,2,i,1,j]$	[0,1]	[0,1]	[0,1]	good	good
KJId	$[k,1,i,0] \rightarrow [k,2,j,1,i]$	[0,1]	[0,1]	[0,1]	good	good
IKJ	$[i,0,k,0] \rightarrow [i,0,k,1,j]$	[0,0,0,1]	[-3,0,0,1]	[-3,0,0,1]	bad	bad
IJK	$[i,0,k,1] \rightarrow [i,0,j,0,k]$	[0,0,+,-1]	[-3,0,0+,-1]	[-3,0,0+,-1,-1]	bad	bad
JKI	$[k,2,i] \rightarrow [j,0,k,i]$	[+,-2]	[0+,-2]	[0+,-1,-2]	bad	bad
JIK	$[k,2,i] \rightarrow [j,0,i,k]$	[+,-2]	[0+,-2]	[0+,-1,2]	bad	bad

Table 5: Predictions made by our algorithm for Cholesky decomposition

checks are neither necessary nor sufficient for a loop to be able to carry interprocessor dependences. The proposed strategy for moving loops is just a heuristic that works well on stencil computations; it isn't clear that it is valid for loops such as linear algebra kernels, and it makes no predictions for block-cyclic decompositions.

Figure 6 shows some of these problems. Loops identified by [7] are marked as `do*`. In the *False Positives* column, loops are marked as do-across even though there are no interprocessor dependences. In the False Negatives column, no loops are marked as do-across even though there are interprocessor dependences. These examples are designed to be demonstrative rather than realistic. There may not be any realistic stencil computations that demonstrate the problems with the above definition. But for non-stencil computations, there are some real codes on which the problems are manifested. For Cholesky decomposition (Figure 3) with the first dimension distributed, it identifies all 3 loops as being "cross-processor", providing no guidance.

In [5], the interaction between placement and scheduling is discussed, and the point is made that the two issues should be decided simultaneously, rather than one after the other. We agree in theory, but believe it is premature to integrate the two decisions until a better understanding of their interactions is achieved (which we hope to advance in this paper).

In Feautrier's work [3], the schedule (i.e., the time mapping) is selected first. This schedule is selected [?], so as to minimize execution time on a PRAM with an infinite number of processors. The resulting schedule is equivalent to sequential loop nests containing parallel loop nests. A global barrier/synchronization is assumed between each iteration of the sequential loops. Once the schedule is selected, a space mapping function from iterations to virtual processors is selected [4] so as to minimize the total communication volume. The space mapping is constrained so that any two iterations executed in parallel according to the schedule must execute on different virtual processors. A folding function (e.g., block or cyclic) is used to map virtual to physical processors.

Feautrier's approach works reasonably well assuming that only barrier synchronization will be used and that computation and communication can be overlapped. If a cyclic folding function is used, using finer grained synchronization and a machine model in which computation and communication can be overlapped will not give any improvement. If a block folding function is used, some improvement may be seen, but his results will generally not be optimal. In the case of the rectangular stencil pattern in Figure 1, he will choose a time mapping of $i+j$: a wavefront going from the lower left corner to the upper right corner. Each processor will start up $B^2/2+l$ after the processor to it's left (where B is the block size and l is the latency), leading to an execution time of $nB+(p-1)(B^2/2+l)$, compared with a total execution time of $nB+(p-1)(B+l)$ for the pipelined time mapping of $\{[i,j] \rightarrow [i,j]\}$.

False Positives	False Negatives

<pre>Decomposition T(N)
real A(N)
Align A(j) with T(j)
Distribute T(block)
do* i = 1 to n
  do* j = i+1 to n
      A(i) = A(i) + 1
      A(j) = A(j) - 1</pre>
	<pre>Decomposition T(N)
	real A(N,N), B(N,N)
	Align A(i,:) with T(i)
	Align B(i,:) with T(i)
	Distribute T(block)
	do i = 1 to n
	  do j = 1 to n
	    A(i,j) = B(i+1,j-1)
	    B(i+1,j) = A(i,j)</pre>

<pre>Decomposition T(N)
real A(N), B(N)
Align A(i) with T(i)
Align B(i) with T(i+1)
Distribute T(block)
do* i = 1 to n
      A(i) = ...
      B(i) = A(i+1)</pre>
	<pre>Decomposition T(N)
	real A(N,N), B(N,N)
	Align A(i,:) with T(i)
	Align B(i,:) with T(i+1)
	Distribute T(block)
	do i = 1 to n
	  do j = 1 to n
	    A(i,j) = B(i,j-1)
	    B(i,j) = A(i,j)</pre>

Figure 6: Errors made by the Fortran-D "cross-processor" identification algorithm

8 Conclusion

We have shown that finding appropriate reordering transformations can be quite important in minimizing idle processor time. Previous work in the Fortran-D system works well in practice on stencil computations, although the algorithms can be spoofed into making wrong choices. For linear algebra kernels, which typically involve imperfectly nested loops with triangular loop bounds, the situation is much more complicated. The Fortran-D strategies do fail in practice for cyclic decompositions and give no help for block-cyclic decompositions. The techniques we describe here are more accurate and consider subtler details, such as skewed iteration spaces.

Other performance characteristics of the system, such as local cache behavior, may dictate other constraints on a "good execution order" for the transformed program. Thus, it is important to have a method of evaluating a potential execution order that can incorporate multiple performance criteria, for example to generate a execution order that has good local cache behavior and minimizes processor idle time. The methods we have described here can be combined with those in [8] to consider multiple performance criteria.

In the final paper, we intend to report results from analyzing additional programs and techniques for analyzing when cyclic computations will exhibit pipelined behavior and validate our theories in either the SUIF or the Fortran-D compiler.

References

[1] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *ACM '93 Conf. on Programming Language Design and Implementation*, June 1993.

[2] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part II, Multidimensional time. *Int. J. of Parallel Programming*, 21(6), Dec 1992. Postscript available as pub.ibp.fr:ibp/reports/masi.92/28.ps.Z.

[3] Paul Feautrier. Compiling for massively parallel architectures: A perspective. In *7th Workshop on Algorithms and Parallel VLSI Architectures*, Leuven, August 1994. Elsevier. to appear.

[4] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, (94), 1994. to appear.

[5] J. Ferrante, G. R. Gao, S. Midkiff, and E. Schonberg. A critial survey of automatic data partitioning. Technical report, IBM T.J. Watson Research Center, October 1992.

[6] High Performance Fortran Forum. High performance fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, 1992. Revised May, 1993.

[7] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for FORTRAN D on MIMD distributed memory machines. In *Supercomputing '91*, November 1991.

[8] Wayne Kelly and William Pugh. Determining schedules based on performance estimation. Technical Report CS-TR-3108, Dept. of Computer Science, University of Maryland, College Park, July 1993. to appear in Parallel Processing Letters (1994).

[9] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-3193, Dept. of Computer Science, University of Maryland, College Park, April 1993.

[10] Amy W. Lim and Monica S. Lam. Communication-free parallelization via affine transformations. In *Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, Cornell, New York, August 1994.

[11] John M. Crummy Semma Hiranandani, Ken Kennedy and Ajay Sethi. Advanced compilation techniques for fortran d. Technical Report CRPC-TR93338, Center for Research on Parallel Computation, Rice Univers ity, October 1993.