UMIACS-TR-97-02 April, 1997
CS-TR-3737

Iteration Space Slicing and Its Application to Communication Optimization

William Pugh
pugh@cs.umd.edu
Inst. for Advanced Computer Studies
Dept. of Computer Science

Evan Rosser
ejr@cs.umd.edu
Dept. of Computer Science

Univ. of Maryland, College Park, MD 20742

Abstract

Program slicing is an analysis that answers questions such as "Which statements might affect the computation of variable v at statement s?" or "Which statements depend on the value of v computed in statement s?". The answers computed by program slicing are generally a set of statements. We introduce the idea of *iteration spacing slicing*: we refine program slicing to ask questions such as "Which iterations of which statements might effect the computation in iterations I of statement s?" or "Which iterations of which statements depend on the value computed by iterations I of statement s?". One application of this general-purpose technique is optimization of interprocessor communication in data-parallel compilers. For example, we can separate a code fragment into 1) those iterations that must be done before a send, 2) those iterations that don't need to be done before a send and don't depend on non-local data and 3), those iterations that depend on non-local data. We examine applications of iteration space slicing to communication optimizations in parallel executions of programs such as stencil computations and block-cyclic Gaussian elimination with partial pivoting.

Keywords: program slicing, communication optimization, message coalescing, latency tolerance

This work is supported by an NSF PYI grant CCR-9157384 and by a Packard Fellowship.

Iteration Space Slicing and Its Application to Communication Optimization

William Pugh
Institute for Advanced Computer Studies
Department of Computer Science

Evan Rosser
Department of Computer Science

University of Maryland, College Park, MD 20742
{pugh,ejr}@cs.umd.edu

Abstract

Program slicing is an analysis that answers questions such as "Which statements might affect the computation of variable v at statement s?" or "Which statements depend on the value of v computed in statement s?". The answers computed by program slicing are generally a set of statements. We introduce the idea of *iteration spacing slicing*: we refine program slicing to ask questions such as "Which iterations of which statements might effect the computation in iterations I of statement s?" or "Which iterations of which statements depend on the value computed by iterations I of statement s?". One application of this general-purpose technique is optimization of interprocessor communication in data-parallel compilers. For example, we can separate a code fragment into 1) those iterations that must be done before a send, 2) those iterations that don't need to be done before a send and don't depend on non-local data and 3), those iterations that depend on non-local data. We examine applications of iteration space slicing to communication optimizations in parallel executions of programs such as stencil computations and block-cyclic Gaussian elimination with partial pivoting.

Keywords: program slicing, communication optimization, message coalescing, latency tolerance

1 Introduction

Program slicing [Wei84] is an analysis that answers questions such as "Which statements might affect the computation of variable v at statement s?" or "Which statements depend on the value of v computed in statement s?". The answers computed by program slicing are generally a set of statements, and the variables considered are generally either scalars or entire arrays. We introduce the idea of *iteration spacing slicing*: we refine program slicing to ask questions such as "Which iterations of which statements might affect the value of elements $1:10$ of array A in iterations I of statement s?" or "Which iterations of which statements depend on the value computed by iterations I of statement s?". In addition, we can specify any subset of an array to be the variable of interest, rather than treating any assignment to the array as an assignment to every location. The slice can be executed to produce the same values for those elements that the full unsliced computation would produce.

We can produce an iteration space slice for programs where loop bounds and array subscripts are affine functions of outer loop bounds and symbolic constants. Iteration set descriptions can be very specific, such as $\{[k,i] : 1 \leq k < i \leq n\}$, and dependences can be described using relations such as $\{[k,i,i] \rightarrow [i,i'] : 1 \leq k < i < i' \leq n\}$. These sets and relations can be represented and manipulated using the Omega library [KMP+95].

Iteration space slicing is not simply a new kind of dependence analysis technique. Data dependence analysis finds pairs of statement instances which access the same memory, and that information can be used to analyze communication or prove legality of transformations. Iteration space slicing takes dependence information as input to find *all* statement instances from a given loop nest which must be executed to produce the correct values for the specified array elements. We can think of the slice as following chains of dependences (i.e. transitive dependences) to reach all statement instances which can affect the result. For example, while dependence information by itself can identify the endpoints of an interprocessor communication, slicing can identify everything that has to be done before that communication can take place.

The key step in calculating an iteration space slice is to calculate the transitive closure of the data dependence graph of the program [KPRS96]; the transitive dependences then are applied to the iterations of interest to produce the slice. Instead of edges being between statements in the graph, edges are actually between iterations of the statements; therefore, we need to compute the transitive closure of an infinite graph, because the iteration spaces may have symbolic bounds. In previous work [KPRS96], we presented techniques for doing this and have implemented them within the Omega library. Computing iteration space slices that are exact in all cases is impossible (as discussed in Section 3 and [KPRS96], there is a reduction to an undecidable problem). However, we can compute upper or lower bounds on a slice, as required. Alternative representations that would be cruder but faster could also be used, although we do not

To appear at the 11th ACM International Conference on Supercomputing, Vienna, Austria, July 1997.

examine them in this paper.

Iteration space slicing is primarily of interest in compiling scientific programs that manipulate arrays, since for these programs the dependencies tend to be such that an iteration space slice might be much smaller than a standard program slice. Slicing provides a framework from which to build compiler optimizations; each optimization that builds upon it chooses what portion of the iteration space to slice, what slices to take, and in what order to execute result.

Because it extracts some subset of the statements in a loop body and a subset of the loop nest's iterations, iteration space slicing can be thought of as a finer-grained approach than many existing transformations, which might act on all statements in a loop, or all iterations of an individual statement. A slice includes all and only those statement instances which must be executed to produce the correct output (up to the precision of the transitive closure computation). It is a data-driven technique in that an optimizer does not work with the loop structure, but instead specifies which data should be computed.

One example of a compiler analysis which can use iteration space slicing is optimizing communication generated by data-parallel compilers for message-passing machines. Given the distribution of data to processors, we can first use dependence analysis to identify the data needed in interprocessor communications. Then, we can slice programs with respect to array elements sent or received, producing the set of computations that must be performed before an interprocessor communication can be sent, and the set of computations that depend on receiving an interprocessor communication. These slices give us great flexibility in reordering communication, since they preserve all the necessary dependences, but exclude computations that don't affect the communication. We explore using this information in the following three ways.

Enabling parallelization of fused loops

Loop fusion is merging the loop structure of two loops to generate a single, *fused* loop which performs the work of both original loops. This transformation can reduce loop overhead, enable scalar replacement and improve cache performance. However, fusing two loops, neither of which carries a dependence, can create a loop-carried dependence which would prevent parallelization of the fused loop (as shown in Figures 1a and 1b). By slicing out the iterations that are involved in interprocessor loop-carried dependences, we can create a large code fragment in which there are no interprocessor loop-carried dependences and loop fusion can be applied profitably. In Figures 1c and 1d, parallel code is shown for a block distribution of `a` and `b`, where `lb` and `ub` denote the boundaries of the array region owned by the local processor. Enabling parallelization of fused loops is discussed in [MA97], and they provide a solution which only applies to programs with constant dependence distances.

Tolerating message latency

In order to hide message latency, compilers try to overlap computation with communication. Consider a loop body (possibly containing inner loops), which may contain interprocessor dependences which are loop-independent with respect to the outer loop. We can divide a loop body into three partitions: 1) those iterations which must be done before some send; 2) those iterations which do not depend on communication; and 3) those iterations which depend on some receive. When those partitions are disjoint, we can execute them in the above order, doing the send between the first two sets and the receive between the second and third. Some of the iterations in the first set may not directly write data which is needed off-processor, but are included because such a write statement transitively depends on them. By doing sends earlier and receives later, we transform programs so that they can tolerate more message latency. This also allows computations to run in a more loosely-coupled fashion, so that when one processor runs slightly ahead of the others, it is not held back. Special cases of this optimization have been discussed in [KMR90, HKT93, AKMC94].

Message aggregation

Message coalescing (or aggregation) is an important transformation on many machines where the start-up time for a message is significant. However, message aggregation exacerbates message latency problems: if we coalesce messages, data is not sent as soon as it is ready, because it is held until there is more to send. In addition, a naive approach simply delays sending the data until all of the values are ready; in that situation, a program may execute some iterations before the sends that could be moved after the sends, further increasing delays (Figure 2). An iteration space slice can extract those computations which *must* be done before the send, and postpone the rest, which enables profitable message coalescing in situations in which it would otherwise introduce significant additional latency. A manual application of this optimization for block-cyclic Gaussian elimination was discussed in [HKMCS94] but automation of the transformation was beyond the capabilities of existing transformation systems. We derive both that transformation and a more sophisticated form that produces even better performance.

The profitability of these optimizations depends on many machine-specific factors, such as message startup time, message latency, and ability to overlap computation with communication. In this paper, we describe the necessary analysis and transformation techniques; methods to estimate the profitability of these optimizations on particular machines are beyond the scope of this paper.

The rest of the paper is structured as follows. In Section 2, we describe the representations we use for iteration sets and dependence relations. In Section 3, we describe the transitive closure of relations, which can be used to compute iteration space slices. In Section 4, we describe the techniques to compute a slice. Section 5 describes applications of the technique to Gaussian elimination. Finally, we present experimental results in Section 6, related work in Section 7, and a conclusion in Section 8.

2 Iteration Sets and Dependences

In order to compute slices which are as small as possible, we need accurate data dependence information. In particular, slicing depends on the ability to identify which individual iterations are involved in a dependence. Most of the previous work on program transformations in loops uses data dependence directions and differences to summarize dependences between array references. In many cases, these abstractions are too imprecise to use for slicing. Array dataflow methods

```
for i = 1 to n do
  a(i) = a(i) + z(i)
for i = 2 to n do
  b(i) = a(i) + a(i-1)
```

(a) Original code

```
a(1) = a(1) + z(1)
for i = 2 to n do
  a(i) = a(i) + z(i)
  b(i) = a(i) + a(i-1)
```

(b) After loop fusion (with loop carried dependence)

```
for i = lb to ub do
  a(i) = a(i) + z(i)
  if i == ub && ub < n then send a(i)
  if i == lb && lb >= 2 receive a(i-1)
  if i >= 2 then
    b(i) = a(i) + a(i-1)
```

(c) Fused and made SPMD

```
if ub < n then
  a(ub) = a(ub) + z(ub)
  send a(ub)
if lb < ub      then a(lb) = a(lb) + z(lb)
for i = lb+1 to ub-1 do
  a(i) = a(i) + z(i)
  b(i) = a(i) + a(i-1)
if lb < ub < n  then b(ub) = a(ub) + a(ub-1)
if ub == n      then a(n) = a(n) + z(n)
if lb < ub == n then b(n) = a(n) + a(n-1)
if 2 <= lb then
  receive a(lb-1)
  b(lb) = a(lb) + a(lb-1)
```

(d) Fused, made SPMD and sliced

Figure 1: Using slicing to enable loop fusion

Original Code	Naive aggregation	Using slicing

`X = ...`
`Send X`
`Y = f(X)`
`Z = g(X)`
`Send Z`
	`X = ...`
	`Y = f(X)`
	`Z = g(X)`
	`Send X and Z`
		`X = ...`
		`Z = g(X)`
		`Send X and Z`
		`Y = f(X)`

Figure 2: Using slicing for message aggregation

```
    for k = 1 to n-1 do
      for i = k+1 to n do
S1      a(i,k) = a(i,k) / a(k,k)
        for j = k+1 to n do
S2        a(i,j) += - a(k,j)*a(i,k)
```

Figure 3: Gaussian elimination without pivoting

[Ros90, TP92, PEH+93, Li92] provide information required to support array privatization, but don't provide information on exactly which iteration produced or needs a value. In order to get the iteration-level detail we require, we describe dependences using integer tuple relations and describe iterations sets using integer tuple sets.

2.1 Integer Tuple Relations and Sets

An integer k-tuple is simply a point in $\mathcal{Z}^k$. A *tuple relation* is a mapping from tuples to tuples. The relations may involve free variables such as n in the following example: $\{ [i] \rightarrow [i+1] \mid 1 \leq i < n \}$. These free variables correspond to symbolic constants or parameters in the source program.

Tuple relations and sets are represented using the Omega library [Pug92, PW95, PW93, KMP+95] which is a collection of routines for manipulating affine constraints over integer variables. The Omega library allows tuple relations and sets to be manipulated using operations such as intersection, union, inverse, composition, and difference.

For example, for the simple form of Gaussian elimination (without pivoting) shown in Figure 3, the sets of iterations of statements S_1 and S_2 can be described as $I_1 = \{[k,i] : 1 \leq k < i \leq n\}$ and $I_2 = \{[k,i,j] : 1 \leq k < i,j \leq n\}$. Dependences are naturally described as relations. For the Gaussian example, the dependences are:

$$
\begin{array}{rl}
D_{1\rightarrow 2} = & \{[k,i] \rightarrow [k,i,j] : 1 \leq k < i,j \leq n\} \\
D_{2\rightarrow 1} = & \{[k,i,j] \rightarrow [j,i] : 1 \leq k < j < i \leq n\} \\
 & \cup \{[k,i,i] \rightarrow [i,i'] : 1 \leq k < i < i' \leq n\} \\
D_{2\rightarrow 2} = & \{[k,i,j] \rightarrow [j',i,j] : 1 \leq k < j' < i,j \leq n\} \\
 & \cup \{[k,i,j] \rightarrow [i,i',j] : 1 \leq k < i < j,i' \leq n\} \\
 & \cup \{[k,i,j] \rightarrow [j,i,j'] : 1 \leq k < j < i,j' \leq n\}
\end{array}
$$

Our assumption that different execution instances of a statement can be usefully described as an iteration set depends on the control flow of the program consisting of well structured loops and conditions. This is not true in many general purpose codes, but is fairly accurate for many scientific numerical applications.

3 Transitive Closure

In addition to the fairly standard operations supported within the Omega library, we also provide a transitive closure operator [KPRS96]. Given a tuple relation F, its transitive closure is the least fixed point of:

$$(x \rightarrow z) \in F^* \Leftrightarrow x = z \vee \exists y \text{ s.t. } (x \rightarrow y) \in F \wedge (y \rightarrow z) \in F^*$$

and its positive transitive closure is the least fixed point of:

$$\begin{aligned}(x \rightarrow z) \in F^{+} \quad & \Leftrightarrow (x \rightarrow z) \in F \\ & \vee \; \exists y \text{ s.t. } (x \rightarrow y) \in F \wedge (y \rightarrow z) \in F^{+}\end{aligned}$$

We can use the transitive closure operation to compute the transitive dependences of a program, which in turn are used to determine which iterations of which statements need to be included in a slice.

Unfortunately, the exact transitive closure of an affine integer tuple relation may not be affine; in fact, it is not computable in the general case. We can encode multiplication using transitive closure (and supporting multiplication allows us to pose undecidable questions):

$$\{[x, y] \rightarrow [x+1, y+z]\}^{*}$$

is equivalent to:

$$\{[x, y] \rightarrow [x', y+z(x'-x)] \mid x \leq x'\}$$

When our methods do not give us an exact answer, they give us both an upper and lower bound on the true transitive closure. However, in practice our techniques frequently do give us the exact answer. In [KPRS96] we found that in 2000 code fragments from our problem domain, we could compute the exact transitive dependences in 99% of the cases. For example, for the Gaussian elimination code fragment, we calculate the exact transitive dependences as:

$$\begin{aligned}
D^{+}_{1\rightarrow 1} = \quad & \{[k,i] \rightarrow [k',i'] : 1 \leq k < i \leq k' < i' \leq n\} \\
& \cup \{[k,i] \rightarrow [k',i] : 1 \leq k < k' < i \leq n\} \\
D^{+}_{1\rightarrow 2} = \quad & \{[k,i] \rightarrow [k',i,j] : 1 \leq k \leq k' < i,j \leq n\} \\
& \cup \{[k,i] \rightarrow [k',i',j] : 1 \leq k < i \leq k' < i',j \leq n\} \\
D^{+}_{2\rightarrow 1} = \quad & \{[k,i,j] \rightarrow [k',i'] : 1 \leq k < i,j \leq k' < i' \leq n\} \\
& \cup \{[k,i,j] \rightarrow [k',i] : 1 \leq k < j \leq k' < i \leq n\} \\
D^{+}_{2\rightarrow 2} = \quad & \{[k,i,j] \rightarrow [k',i,j] : 1 \leq k < k' < i,j \leq n\} \\
& \cup \{[k,i,j] \rightarrow [k',i,j'] : 1 \leq k < j \leq k' < i,j' \leq n\} \\
& \cup \{[k,i,j] \rightarrow [k',i',j] : 1 \leq k < i \leq k' < j,i' \leq n\} \\
& \cup \{[k,i,j] \rightarrow [k',i',j'] : 1 \leq k < i,j \leq k' < i',j' \leq n\}
\end{aligned}$$

4 Iteration space slicing

In traditional program slicing, given the data and control flow edges, the set of statements that are reachable via the transitive closure of the dependence edges (in a forwards or backwards direction, depending on the application) must be computed; the transitive closure of a finite graph can be easily solved. In our application, where we find iterations of statements, the transitive closure of the dependences is a much more complicated problem, as described in Section 3. Once we have the transitive closure of the dependences, computing the iterations reachable from the designated iterations is a straightforward application of the transitive dependences to the iterations of interest.

In traditional program slicing, the final significant issue is determining how to construct an executable slice [Wei84, Tip95]. Within our domain, this problem is very simple because we assume well-structured control flow. Given a set of iteration spaces, we can use techniques [KPR95] implemented within the Omega library to generate efficient code that enumerates those iteration spaces, executing the selected iterations in the same order with respect to other selected iterations.

As an example of an iteration space slice, consider Gaussian elimination with a block-cyclic distribution of the second dimension of the matrix. First, we find iterations which produce data which will be sent to other processors. With this distribution, only one processor sends data within an iteration of the outer loop; the role of the sending processor rotates from one processor the next. Let *width* be the size of a cyclic block, and *lb* be the lower bound on the columns in the processor's current block of the matrix.[1] When a particular processor p becomes the sender, it will send columns $lb \ldots lb + width - 1$ to all other processors; these individual columns are candidates for message aggregation. To avoid partially serializing the computation, we want to perform only the necessary computations before the send. These values are computed during iterations

$$S_1 = I_1 \cap \{[\mathrm{lb} : \mathrm{lb} + \mathrm{width} - 1, *]\}$$

If we compute a backwards slice of these iterations with respect to the transitive dependences, we get:

$$\begin{aligned}
P_1 \quad & = (D^{*}_{1\rightarrow 1})^{-1}(S_1) \\
& = I_1 \cap \{[k,i] : \mathrm{lb} < n \wedge k < \mathrm{lb} + \mathrm{width}\} \\
P_2 \quad & = (D^{*}_{2\rightarrow 1})^{-1}(S_1) \\
& = I_2 \cap \{[k,i,j] : \mathrm{lb} < n \wedge j < n \wedge j < \mathrm{lb} + \mathrm{width}\}
\end{aligned}$$

We are not interested in the entire sets P_1 and P_2, since they include every iteration (back to the beginning of the program) on which S_1 depends. Instead, we want to restrict our attention to the current set of iterations of the k loop ($\mathrm{lb} \leq k < \mathrm{lb} + \mathrm{width}$). (We discuss extending this set in Section 5.4.) The iterations of the slice that occur while $\mathrm{lb} \leq k < \mathrm{lb} + \mathrm{width}$ are:

$$\begin{aligned}
Q_1 \quad & = P_1 \cap \{[\mathrm{lb} : \mathrm{lb} + \mathrm{width} - 1, *]\} \\
& = \{[k,i] : 1 \leq \mathrm{lb} \leq k < i \leq n \wedge k < \mathrm{lb} + \mathrm{width}\} \\
Q_2 \quad & = P_2 \cap \{[\mathrm{lb} : \mathrm{lb} + \mathrm{width} - 1, *, *]\} \\
& = \{[k,i,j] : 1 \leq \mathrm{lb} \leq k < i \leq n \wedge k < j < n \wedge \\
& \qquad j < \mathrm{lb} + \mathrm{width}\}
\end{aligned}$$

The computations for $\mathrm{lb} \leq k < \mathrm{lb} + \mathrm{width}$ that are not part of the pre-send slice are:

$$\begin{aligned}
R_1 \quad & = (I_1 - P_1) \cap \{[\mathrm{lb} : \mathrm{lb} + \mathrm{width} - 1, *]\} = \emptyset \\
R_2 \quad & = (I_2 - P_2) \cap \{[\mathrm{lb} : \mathrm{lb} + \mathrm{width} - 1, *, *]\} \\
& = \{[k,i,n] : 1 \leq \mathrm{lb} \leq k < i \leq n \wedge k < \mathrm{lb} + \mathrm{width}\} \\
& \quad \cup \{[k,i,j] : 1 \leq \mathrm{lb} \leq k < i \leq n \wedge k < \mathrm{lb} + \mathrm{width} \leq j < n\}
\end{aligned}$$

Using the code generation facilities of the Omega library, we can generate code for the slice (Q_1,Q_2) that must be done before the send, and code for the "complement" of the slice (the iterations that can be done after the send (R_1,R_2)). Within both the slice and the complement, the iterations are performed in their original order (although reordering transformations could be applied to the individual slices). Figure 4 represents the code generated by the Omega library.

Note that the code in Figure 4 incorporates one subtle feature that at first seems confusing. When $n \leq$ `lb+width`-1, we do not perform column updates of the last column before doing the send; since column n is never sent, updating it is never part of a send slice. This is a valid optimization, which is in fact required by the rules of our transformation. In practice, it will not affect the performance but it comes from having a formally derived transformation.

[1] *lb* may be symbolic, which allows us to apply this technique when the number of processors is unknown.

```
for k = 1 to n do
  p[k] = indexOfMaxAbs(a[k:n,k])
  for j = k to n do
    swap(a[k,j], a[p[k],j])
  for i = k+1 to n do
    a[i,k] = a[i,k] / a[k,k]
    for j = k+1 to n do
      a[i,j] = a[i,j] - a[k,j] * a[i,k]
```

a) Gaussian elimination with partial pivoting

```
for k = 1 to n do
  p[k] = indexOfMaxAbs(a[k:n,k])
  for j = k to n do
    swap(a[k,j], a[p[k],j])
  for i = k+1 to n do
    a[i,k] = a[i,k] / a[k,k]
  send comm
  receive comm
  for i = k+1 to n do
    for j = k+1 to n do
      a[i,j] = a[i,j] - a[k,j] * a[i,k]
```

b) with communication added

```
for kB = 1 to n by width do
  for k = kB to min(kB+width,n) do
    p[k] = indexOfMaxAbs(a[k:n,k])
    for j = k to n do
      swap(a[k,j], a[p[k],j])
    for i = k+1 to n do
      a[i,k] = a[i,k] / a[k,k]
    send comm
    receive comm
    for i = k+1 to n do
      for j = k+1 to n do
        a[i,j] = a[i,j] - a[k,j] * a[i,k]
```

c) blocked according to sender

```
for kB = 1 to n by width do
  /* Only if I am not the sender */
    receive columns kB..min(kB+width,n) of a and p
  for k = kB to min(kB+width,n) do
    p[k] = indexOfMaxAbs(a[k:n,k])
    for j = k to n do
      swap(a[k,j], a[p[k],j])
    for i = k+1 to n do
      a[i,k] = a[i,k] / a[k,k]
    for i = k+1 to n do
      for j = k+1 to n do
        a[i,j] = a[i,j] - a[k,j] * a[i,k]
  /* Only if I am the sender */
    send columns kB..min(kB+width,n) of a and p
```

d) Naive message aggregation

```
foreach sender,
  do timeStep intersection slice
  send comm
  receive comm
  do timeStep - slice
```

e) Simple schema for improving message aggregation

```
foreach sender1,sender2,
  do timeStep1 intersection slice1
  send comm1
  receive comm1
  do timeStep1 - slice1
  do timeStep2 intersection slice2
  send comm2
  receive comm2
  do timeStep2 - slice2
```

f) Resulting of unrolling (e)

```
foreach sender1,sender2,
  do timeStep1 intersection slice1
  send comm1
  receive comm1
  do (timeStep1 - slice1) intersection slice2
  do timeStep2 intersection slice2
  send comm2
  do (timeStep1 - slice1) - slice2
  receive comm2
  do timeStep2 - slice2
```

g) Slicing (f) to send message quicker

```
foreach sender
  do timeStep_curr intersection slice_curr
  send comm
  do timeStep_prev - slice_prev - slice_curr
  receive comm
  do (timeStep_curr - slice_curr) intersection slice_next
do timeStep_last - slice_last
```

h) Pipelining (g)

Figure 5: Forms and schemas for transforming Gaussian elimination

```
for(k = lb; k <= min(n-1,lb+width-1); k++)
  for(i = k+1; i <= n; i++) {
    a[i,k] = a[i,k] / a[k,k]
    for(j = k+1; j <= min(lb+width-1,n-1); j++)
      a[i,j] = a[i,j] - a[k,j] * a[i,k]
  }
send columns
for(k = lb; k <= min(n-1,lb+width-1); k++)
  for(i = k+1; i <= n; i++) {
    for(j = width+lb; j < n; j++)
      a[i,j] = a[i,j] - a[k,j] * a[i,k]
    a[i,n] = a[i,n] - a[k,n] * a[i,k]
  }
```

Figure 4: Sliced generated for Gaussian elimination without pivoting

5 An application of slicing to Gaussian elimination

In Figure 5a, we show a form of Gaussian elimination with partial pivoting that will be the basis of our transformation. Note that we have assumed the existence of a primitive operation to find the pivot of each column. If the source used a loop containing a conditional to find the pivot, the dependence analysis might be more difficult.

5.1 Standard parallel form for Gaussian elimination

In Figure 5b, we sketch the form of a parallel form of Gaussian elimination, where the matrix is distributed by columns. One minor optimization that has been applied here is to distribute the `i` loop so as to allow the pivot column to be sent earlier. This is the code that would be generated by most parallelizing compilers for this problem.

In all parallel codes, we assume that the transformations to limit the computations to those executed on each processor, and to limit the sends and receives to just the appropriate processors, are performed in a later step.

5.2 Naive Aggregation

With a block-cyclic distribution, it appears attractive [HKMCS94] to coalesce the communication of the adjacent columns that belong to one processor.

A first approach might be as follows: we first block the k loop by the width of the cyclic blocks, so that in each block, a single processor owns the elements being communicated, as shown in Figure 5c. Unfortunately, a dependence carried by the k loop prevents distribution of the inner-k loop, which would allow us to place the communication effectively. To aggregate the communication, we are reduced to simply moving the communication out of the inner k-loop, as shown in Figure 5d. Since within each k-block, only a single processor is sending data, this is guaranteed to be safe and deadlock-free. However, it serializes a substantial part of the computation: the sending processor performs all the computations of a single iteration of kB before sending, although the send itself does not require it. Traditional transformations cannot extract only the necessary portion of the second i loop.

5.3 Slicing within a time step

To get better performance from coalescing, we need to perform only those computations which *must* be done before the send, and delay everything else until after the communication. To extract only the necessary computations, we need to use slicing.

Once again, we block the outer `k` loop so that within each block, only a single processor is computing values that are needed on other processors. In the same block of the `k` loop, the processor will also compute values which will not be sent; we want to first compute the values that will be communicated, and delay the execution of those that won't be.

We produce an iteration space slice using as the criteria the region to be sent within a block of k. By the definition of an iteration space slice, it contains every computation which can affect the sent data, and excludes the computations which made the naive approach unprofitable. Separate iteration space slices are computed to perform the computations which depend upon received values, and those which can be done at any point in the block of `k`. The result is shown in Figure 5e. This slicing gives us code similar to that in Figure 4 (although that code doesn't include pivoting). This is equivalent to an optimization proposed in [HKMCS94].

5.4 Slicing across multiple time steps

Slicing the code within a time step permits us to coalesce messages without a huge penalty, and depending on machine details, may improve performance by itself. However, it does not hide the latency of that communication. Since all the processors start the execution of the next iteration of the `lb` loop at about the same time, the receiving processors must wait for the message from the sending processor. Even if we send the message as quickly as possible, there will be a delay. We can further improve the code by making sure that the sending processor gets a head start on each time step of `k`, so that the message will be waiting for receiving processors when they need it, by postponing operations from the previous time step.

One approach is to unroll the loop, exposing a larger portion of the iteration space to be sliced. Figure 5f shows the effect of unrolling the loop in Figure 5e. We can see that between receiving communications 1 and sending communications 2, we might be doing more work than needed. We can use slicing to delay unneeded work until after sending communications 2, as shown in Figure 5g. This improves the latency tolerance within the outer loop, but doesn't help between one iteration and the next. By using techniques similar to those used for software pipelining, we can get the effects of unrolling the loop many times without actually unrolling the loop; this gives us the schema shown in 5h. The transformed code for Gaussian elimination is too large to include here, but can be found at `http://www.cs.umd.edu/projects/omega/slicing`. This same scheme could be applied directly to Cholesky decomposition and to QR decomposition via Given's rotations.

6 Experimental Results

Experiments were performed on an IBM SP/2 with 16 processors, running AIX 4.1 and using IBM MPI libraries, and using the high-speed switch in interrupt-driven mode. The interrupt-driven mode imposes an additional overhead on each communication, but permits non-blocking communication (when the switch is used in polled mode, no appreciable communication occurs unless both processors are simultaneously executing communication code [SHFG95]). The optimizations we describe are very dependent on having a communication system that supports non-blocking I/O that can be overlapped with computation.

6.1 Gaussian elimination with partial pivoting

We ran three parallel versions of Gaussian elimination with partial pivoting using both a cyclic and block-cyclic distribution. We specified the loop to slice, and performed the loop blocking manually, and then program slices were generated automatically. Communication statements were inserted manually. The first code was a straightforward parallelization with no coalescing (Figure 5c). The second code performed coalescing via slicing a single time step (Figure 5e). The third code corresponds to the pipelined version (Figure 5g).

Two optimizations were hand-applied to all codes (including the sequential version against which speedups were computed). The matrix was transposed from its original order, and the `j` and `i` loops were interchanged in order to achieve better cache performance. Parallel versions scaled very poorly without this transformation. We hope that incorporating our techniques into a data-parallel compiler will allow us to take advantage of a full set of optimizations and improve performance.

We ran the codes for problem sizes 512, and 1024; for cyclic block sizes 1,2, 4, 8, 16, and 32; on 2, 4, 8, 12, and 16 processors. In nearly all cases, the version sliced over multiple time steps outperformed the broadcasting version. Furthermore, the version sliced across multiple time steps was able to hide significantly more latency than the version sliced over one step.

Speedups for some of those problem sizes are presented in Tables 1 and 2 Speedups are computed against a sequential version, with no parallel overhead. Note that in the cyclic(1) case, the version which is sliced within a time step should be identical to the code in with each column is broadcast as soon as it is computed; differences are due to the different loop structure and different communication calls used.

Overall speedups are only fair. We found that toward the end of the execution, as the number of column applications per block of the k loop shrank, the low ratio of computation to communications was a large performance hit. On some smaller problem sizes on larger numbers of processors, the amount of work assigned to each processor was not sufficient to hide latency of communication. In these cases the aggregation without latency tolerance can be detrimental.

7 Related Work

Although there is a large body of work on program slicing [Wei84, Tip95, Kri], we are unaware of any work describing the concept of computing an iteration space slice. However, our work does not address many of the important issues that others have studied, such as complicated or irregular control flow and interprocedural slicing.

Manjikian and Abdelraham [MA97] discussed the problem of loop-fusion creating loop-carried dependences and preventing parallelism. Their solution [MA97] works only in the presence of constant dependence distances and generates code that requires a barrier and will be inefficient if message latency is high. Since their method requires constant dependence distances, it works well in the case of non-periodic stencil computations. However, if the stencil computation is periodic (e.g., the first column is considered to be the right neighbor of the last column), the non-constant distance dependence distance will make it inapplicable. Their paper also discusses a number of issues related performing loop fusion and avoiding cache conflicts while doing so. We do not address those issues here, and might profit from using their methods.

Adve, Koelbel, and Mellor-Crummey [AKMC94] note that splitting off the iterations that are needed before or depend on non-local computations can be beneficial, and discuss how this is done in simple stencil computations such as SOR. Simple cases of this optimization have been discussed or implemented by a number of researchers [KMR90, HKT93]

Hiranandani et.al.[HKMCS94] discuss a number of issues related to the generation of efficient code for block-cyclic data distributions. In their conclusion, they mention that aggregating the sends of each column would decrease the communications costs, and suggest the transformation described in Section 5.3, but admit that the techniques described in the paper are insufficient to enable or derive that transformation. The additional level of optimization across multiple time steps described in Section 5.4 is not discussed.

Adve, Koelbel and Mellor-Crummey [AKMC94] describe an optimization for which "...no parallelizing compiler has attempted the more difficult analysis and reordering of computations needed to deduce the optimization of DGEFA from the original source." This optimization is a restricted case of the transformation described in Section 5.4, for a pure cyclic distribution.

8 Conclusion

We have described iteration space slicing, an extension of program slicing that computes the set of iterations that depend on or influence the events of interest. While this analysis might have a number of uses, we have focused on taking slices of parallel programs with respect to computations that produce or depend on inter-processor communication. In particular, we can use slicing to enable loop fusion, tolerate message latency and allow message coalescing. We have explored several different patterns or schemas for using slicing for these purposes, but we don't claim to have exhausted the set of such schemas, or found the very best schemas.

Our techniques for allowing loop fusion, tolerating latency, and allowing message coalescing encompass the techniques proposed by other researchers, while being more general (for example, by not requiring constant dependence distances). For Gaussian elimination, we have been able to derive sophisticated program transformations from a simple and high-level schema. For the pure cyclic case, we derive code which is equivalent to that described as too difficult for automatic derivation by other researchers. For the block-cyclic case, we have derived a more sophisticated form which incorporates the same message aggregation but also hides the latency which aggregation introduces. Our experimental results with the block-cyclic code show improvements over code without the optimization. We plan to incorporate the technique into an optimizing data-parallel compiler, which

Figure	Program	Cyclic size	2 Proc	4 Proc	8 Proc	12 Proc	16 Proc
5c	Unsliced, Bcast columns uncoalesced	1	2.01	3.53	5.52	6.32	7.43
5e	Sliced within a time step	1	1.99	3.58	5.49	6.13	6.37
5g	Sliced over multiple steps	1	2.03	3.75	6.47	8.00	8.44
5c	Unsliced, Bcast columns uncoalesced	8	2.01	3.59	5.78	6.86	7.71
5e	Sliced within a time step	8	2.03	3.67	5.90	6.87	7.16
5g	Sliced over multiple steps	8	2.09	3.99	7.28	8.99	9.30

Table 1: Speedups for problem size 1024

Figure	Program	Cyclic size	2 Proc	4 Proc	8 Proc	12 Proc	16 Proc
5c	Unsliced, Bcast columns uncoalesced	1	1.87	3.03	3.93	3.92	4.71
5e	Sliced within a time step	1	1.86	3.20	3.97	3.94	3.60
5g	Sliced over multiple steps	1	1.94	3.41	5.09	5.32	4.59
5c	Unsliced, Bcast columns uncoalesced	8	1.89	2.94	4.14	4.50	4.86
5e	Sliced within a time step	8	1.95	3.17	4.36	4.65	4.43
5g	Sliced over multiple steps	8	2.09	3.77	5.89	5.81	4.99

Table 2: Speedups for problem size 512

we hope will improve the overall speedup of the code.

References

[AKMC94] Vikram S. Adve, Charles Koelbel, and John M. Mellor-Crummey. Performance analysis of data-parallel programs. Technical Report CRPC-TR94404, Center for Research on Parallel Computation, Rice University, May 1994.

[HKMCS94] Seema Hiranandani, Ken Kennedy, John Mellor-Crummey, and Ajay Sethi. Compilation techniques for block-cyclic distributions. In *Proc. of the 1994 International Conference on Supercomputing*, 1994.

[HKT93] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Preliminary experiences with the fortran-d compiler. In *Supercomputing '93*, November 1993.

[KMP+95] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, Dept. of Computer Science, University of Maryland, College Park, March 1995. The Omega library is available from http://www.cs.umd.edu/projects/omega.

[KMR90] Charles Koebel, Piyush Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory machine s. In *Proc. of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1990.

[KPR95] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *The 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Virginia, February 1995.

[KPRS96] Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. Transitive closure of infinite graphs and its applications. *International J. of Parallel Programming*, 24(6):579–598, December 1996.

[Kri] Jens Krinke. Program slicing page. http://www.cs.tu-bs.de/~krinke/Slicing/slicing.html.

[Li92] Zhiyuan Li. Array privatization for parallel execution of loops. In *Proc. of the 1992 International Conference on Supercomputing*, pages 313–322, July 1992.

[MA97] Maraig Manjikian and Tarek SS. Abdelrahman. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems*, February 1997.

[PEH+93] David A. Padua, Rudolf Eigenmann, Jay Hoelflinger, Paul Peterson, Peng Tu, Stephen Weatherford, and Keith Faigin. Polaris: A new-generation parallelizing compiler for mpps. CSRD Rpt. 1306, Dept. of Computer Science, University of Illinois at Urbana-Champaign, June 1993.

[Pug92] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[PW93] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Lecture Notes in Computer Science 768: Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Portland, OR, August 1993. Springer-Verlag.

[PW95] William Pugh and David Wonnacott. Going beyond integer programming with the Omega test to eliminate false data dependences. *IEEE Transactions on Parallel and Distributed Systems*, pages 204–211, February 1995.

[Ros90] Carl Rosene. *Incremental Dependence Analysis*. PhD thesis, Dept. of Computer Science, Rice University, March 1990.

[SHFG95] M. Snir, P Hochschild, D. D. Frye, and K. J. Gildea. The communication software and parallel environment of the ibm sp2. *IBM Systems Journal*, 34(2):205–221, 1995.

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[TP92] Peng Tu and David Padua. Array privatization for shared and distributed memory machines. In *Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, September 1992.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, pages 352–357, July 1984.