International Journal of Parallel Programming, Vol. 28, No. 2, 2000

A Loop Transformation Algorithm for Communication Overlapping

Kazuaki Ishizaki,[1] Hideaki Komatsu,[1] and Toshio Nakatani[1]

Received March 23, 1998; revised April 7, 1999

Overlapping communication with computation is a well-known approach to improving performance. Previous research has focused on optimizations performed by the programmer. This paper presents a compiler algorithm that automatically determines the appropriate loop indices of a given nested loop and applies loop interchange and tiling in order to overlap communication with computation. The algorithm avoids generating redundant communication by providing a framework for combining information on data dependence, communication, and reuse. It also describes a method of generating messages to exchange data between processors for tiled loops on distributed memory machines. The algorithm has been implemented in our High Performance Fortran (HPF) compiler, and experimental results have shown its effectiveness on distributed memory machines, such as the RISC System/6000 Scalable POWERparallel System. This paper also discusses the architectural problems of efficient optimization.

KEY WORDS: compiler; distributed memory machines; overlapping communication with computation; loop transformations.

1. INTRODUCTION

On distributed memory machines, data parallel programming provides an effective way of writing scalable parallel programs. In this model, the user writes message-passing programs that deal with separate address spaces, communication, and synchronization. Therefore, many parallelizing compilers[1–6] have been developed for the Fortran-D and High Performance Fortran (HPF)[7] languages. These compilers make it possible for the user to write programs in the global memory address space without dealing

[1] 1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan.

with the details of interprocessor communication and synchronization. To obtain scalability with parallel compilers, it is important to use message related optimization that combines a series of short messages in a loop into a single long message to reduce the startup overhead of each communication. It is the key optimization in parallelizing compilers for distributed memory machines[8].

Many researchers have proposed methods for reducing execution time by overlapping communication with computation. This hides the communication latency during computation. In previous methods,[9–11] the programmer had to manually insert nonblocking send/receive[12] or PUT/GET primitives in order to use the optimization method of overlapping communication with computation. Therefore, the cited papers do not discuss loop transformations by a compiler for overlapping communication with computation. Various methods of executing a loop in a tiled wavefront method have also been described[13–15] for situations in which the loop contains a true dependence. These methods apply tiling to the loop in order to overlap communication with computation and reduce the synchronization cost on distributed memory machines. The cited papers discuss methods for choosing the best tile size according to the characteristic parameters of the machine; however, they do not discuss methods for applying loop transformations automatically.

This paper presents an algorithm that automatically transforms loops using loop interchange[16, 17] and tiling[17–19] in order to overlap communication with computation. A framework for combining information about data dependence, communication, and reuse to allow automatic application of the loop transformations is proposed. It is expressed in the framework of Banerjee's[16] and Wolfe's[19] unimodular framework. This paper also describes a method of generating message to exchange data between processors for tiled loops on a distributed memory machine. The algorithm has been implemented in our High Performance Fortran (HPF) compiler, and experimental results have shown its effectiveness on distributed memory machines, the RISC System/6000 SP.[20]

Tiling is a loop transformation that a compiler uses to automatically create a blocked nested loop. Tiling divides the iteration space into blocks or tiles of the same size and shape and traverses the tiles to cover the entire iteration space. On a uniprocessor, tiling is used to improve the cache locality by changing the order of data accesses in a loop so that reuse of the elements occurs in the innermost loop.[21] It takes advantage of the benefits of the memory hierarchy. On distributed memory machines, if reuse with nonlocal data accesses occurs in the innermost loop, the performance is greatly degraded. For tiling on distributed memory machines, our algorithm applies loop interchange so that communication by nonlocal data accesses without reuse occurs at the outermost loop. When reuse with nonlocal accesses does not occur in a loop index, our algorithm applies tiling to that loop index in order to overlap communication with computation.

The outline of our algorithm is as follows: First, the algorithm calculates the communication, called the In Set,(2) required when a nested loop is executed in parallel. It then determines the indices of loops in which reuse does not occur for arrays requiring communication, using a combination of data dependence, communication, and reuse information. The algorithm then transforms the determined loop indices by means of loop interchange to perform communication without reuse at the outermost possible nested loop, and applies tiling to the loop indices. Finally, it generates messages to exchange data between processors just before and after the tiled loop. As a result, the communication is split into small pieces based on the tiled loop. The processor can overlap this communication with computation.

The structure of this paper is as follows. Section 2 summarizes related work. Section 3 introduces definitions of some terms used throughout the paper. Section 4 presents the concept behind our algorithm. Section 5 describes our loop transformation algorithm. Section 6 describes a method of generating messages to exchange data between processors. Section 7 gives the performance results that we obtained in our experiments, and discusses the bottlenecks of the system. Finally, Section 8 outlines our conclusions.

2. RELATED WORK

We summarize related work in two categories: overlapping communication with computation, and loop transformations for distributed memory machines.

2.1. Overlapping Communication with Computation

We categorize previous work into the following six areas:

1. Overlapping communication with computation by using pipeline communication in loops where there is a true dependence;(13–15) these loops are tiled by a compiler. The cited papers discuss methods for choosing the best tile size.
2. Overlapping the communication for the next computation with the current computation in loops that require prefetch communication.(9, 10) The cited papers discuss the performance of programs optimized by the programmer.

3. Overlapping communication with computation by transforming the original loops with stencil communication into loops with two separate computation processes, one with nonlocal data and the other with local data.[2, 22] The cited papers discuss methods for generating code in the limited cases of FORALL statements or one-dimensional distributed arrays.
4. Overlapping the communication for selection of the next pivot with computation of matrix elimination in Gaussian Elimination.[21] The cited paper discusses an optimization, to shorten the critical path of the computation, that is performed manually by the programmer.
5. Hiding communication latency and eliminating redundant communication by scheduling sends as early as possible and receives as late as possible.[23] The cited paper discusses a compiler framework for eliminating redundant communications, but does not give any experimental results.
6. Hiding data transfer overhead by pre-loading and post-storing on shared memory processors.[24] The cited paper discusses a static scheduling algorithm in the case that execution times of each task and data transfer are given.

In areas 1–4, the cited papers do not discuss loop transformation algorithms that can be implemented in an automatic parallelization compiler. In areas 1 and 2, our algorithm can apply loop transformations to nested loops in order to overlap communication with computation automatically. The nested loops transformed by our algorithm are the same as in previous methods.[9, 10, 13–15] However, the cited papers do not discuss any algorithms for automatic loop transformation, such as the one implemented in our compiler. Our algorithm makes it possible for communication to be overlapped with computation in languages that do not insert communication primitives explicitly, such as HPF. It can be applied to loops that include operands requiring prefetch and pipeline communication.

2.2. Loop Transformations

Much research[25–27] has been done on automatic loop transformations to improve the performance of nested loops on distributed memory machines. The cited papers focus on a two-phase approach that consists of automatic data distribution and loop transformations to minimize the amount of communication in nested loops, while our algorithm focuses on reducing the execution time in a nested loop by overlapping communication with computation. Therefore, our algorithm can be applied to nested loops transformed by other algorithms.

3. BACKGROUND

In this section, we introduce our loop representation, data dependence vectors, and tiling in order to explain our algorithm.

3.1. Loop Representation

To simplify the discussion, we discuss only perfectly nested loops. We represent a perfectly nested loop, such as that in Fig. 1a, as shown in Fig. 1b. Here, l and r are m-dimensional arrays in an n-nested loop. Each iteration in the loop is identified by a column vector $\mathbf{i} = (i_1, ..., i_n)$. We call this vector the loop index vector, and each element a loop index. Here, i_i is the value of the ith loop index, counting from the outermost to the innermost loop. The lower bound of the loop is denoted by $\mathbf{l} = (l_1, ..., l_n)$, and the upper bound of the loop is denoted by $\mathbf{u} = (u_1, ..., u_n)$. The subscript function $f(\mathbf{i}) = H\mathbf{i} + \mathbf{a} = \mathbf{j}$ maps the loop index vector $\mathbf{i}$ to the array vector $\mathbf{j} = (j_1, ..., j_m)$. All elemental values of the linear transformation matrix H and the vector $\mathbf{a}$ are integers.

The distribution function $g(\mathbf{j}) = \mathbf{p}$ maps the array index vector $\mathbf{j}$ to the processor index vector $\mathbf{p} = (p_1, ..., p_k)$, where k is the rank of the processor configuration. Let P be the set of processor indices for executing the program.

In Fig. 2, n is three, $\mathbf{l}$ is (1, 1, 1), $\mathbf{u}$ is (8, 8, 8), k is one, and P is $\{(0), (1)\}$. The rank m of the array A is two. The mapping function of a reference $A(I, J)$ is $f(\mathbf{i}) = \left(\begin{smallmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{smallmatrix}\right)\mathbf{i}$. The distribution function of the array A is $g(\mathbf{j}) = (\lfloor (j_2 - 1)/4 \rfloor)$.

```
DO i1 = l1, u1
  ...
   DO in = ln, un
     l(i1,..,im) = r(i1,..,im)
   END DO
  ...
END DO
(a) General representation
```

```
DO $\vec{i}$ = $\vec{l}$ to $\vec{u}$
  l($f_l(\vec{i})$) = r($f_r(\vec{i})$)
ENDDO
(b) Our representation
```

Fig. 1. Loop representation.

```
      REAL A(8,8), B(8,8), C(8,8)
*HPF$ PROCESSORS P(2)
*HPF$ DISTRIBUTE (*,BLOCK) onto P :: A, B, C
      DO 10 J = 1, 8
       DO 10 I = 1, 8
        DO 10 K = 1, 8
   10     A(I,J)=A(I,J)+B(I,K)*C(K,J)
```

Fig. 2. An example program.

3.2. Dependence Vector

A dependence vector shows possible execution orders constrained by their data dependence. In our approach, we distinguish between true and anti-data dependencies in order to analyze data dependence exactly. In this paper, an anti dependence vector $\mathbf{d}_a$ shows that an execution order is constrained by anti dependence. A true dependence vector $\mathbf{d}_t$ shows that an execution order is constrained by true dependence.(28) We define D, which is the set consisting of $\mathbf{d}_a$ and $\mathbf{d}_t$, as follows:

$$\begin{aligned}
&D = D_a \cup D_t,\\
&D_a = \{\mathbf{d}_a \mid \mathbf{d}_a = \mathbf{i}' - \mathbf{i}, \text{ anti dependence from } \mathbf{i} \text{ to } \mathbf{i}'\},\\
&D_t = \{\mathbf{d}_t \mid \mathbf{d}_t = \mathbf{i}' - \mathbf{i}, \text{ true dependence from } \mathbf{i} \text{ to } \mathbf{i}'\}\\
&\mathbf{d}_a = (d_{a1}, \ldots, d_{an}),\\
&d_{ai} = [d_{ai}^{\min}, d_{ai}^{\max}], \quad d_{ai}^{\min} \in Z \cup -\infty, \quad d_{ai}^{\max} \in Z \cup \infty,\\
&\mathbf{d}_t = (d_{t1}, \ldots, d_{tn}),\\
&d_{ti} = [d_{ti}^{\min}, d_{ti}^{\max}], \quad d_{ti}^{\min} \in Z \cup -\infty, \quad d_{ti}^{\max} \in Z \cup \infty
\end{aligned}$$

3.3. Iteration Space Tiling

In general, tiling decomposes an n-nested loop into a $n+t$-nested loop by adding t inner loops for a fixed number of iterations. This reduces the number of iterations between accesses of the same data, which allows the same data to be kept in the data cache, and hence reduces the number of memory accesses. Thus, on a uniprocessor, tiling is used to improve the data locality. Figure 3 shows the code after tiling of both the loop index variables I and K for the example shown in Fig. 2, using a tile size of 4×4.

4. CONCEPT OF THE ALGORITHM

In this section, we give a general definition of reuse. We then discuss the usage of reuse information for distributed memory machines.

4.1. Types of Reuse

Reuse[21] occurs when the same data is read or written more than once in a loop. Temporal reuse occurs when two references access the same memory location. Spatial reuse occurs when two references access nearby memory locations, such as within the same cache line. A self reuse occurs when a reference in different iterations accesses the same memory location. A group reuse occurs when different references access the same memory location. The following terms[21] are used for these types of reuse. Self-temporal reuse is a reference that accesses the same memory location in different iterations. Self-spatial reuse is a reference that accesses the same cache line in different iterations. Group-temporal reuse is a reference that accesses the same memory location. Group-spatial reuse is a reference that accesses the same cache line.

4.2. Temporal Reuse

On distributed memory machines, $A(I)$ and $A(I+1)$, which are consecutive from a programming language viewpoint, are not necessarily assigned consecutive physical memory locations; this depends on the array distribution. This makes it hard to discuss spatial reuse generally in considering the array distribution. We therefore discuss only temporal reuse in this paper.

4.2.1. Self-Temporal Reuse

First, we consider self-temporal reuse. Self-temporal reuse occurs if two iterations $\mathbf{i}_1$ and $\mathbf{i}_2$ reference the same data element. This occurs whenever $H\mathbf{i}_1+\mathbf{a}=H\mathbf{i}_2+\mathbf{a}$, that is, when $H(\mathbf{i}_1-\mathbf{i}_2)=0$. The solution of this equation is H, called the self-temporal reuse vector R_{ST}. Practically, R_{ST} is the dimension corresponding to the subscript in which the loop index variables do not appear.

On a uniprocessor, choosing some indices from the zero-value of ker H as the innermost loop index allows a reference to exploit self-temporal reuse, because reuse occurs when the value of the loop index variable that does not appear in the subscripts is changed. An example of this is the loop index variable K for the array B in Fig. 2. Tiling is applied to these indices of the loop nest, and the tiled loops are moved to the innermost position to maximize the number of times the processor reads the same elements (from the data cache) within the innermost loop. The key to increasing performance is to improve the cache locality in an innermost loop.

Here, however, we discuss the case in which this method is used on distributed memory machines. If reuse with non-local data access occurs in

```
          REAL A(8,8), B(8,8), C(8,8)
          DO 10 I = 1, 8, 4
           DO 10 K = 1, 8, 4
            DO 10 J = 1, 8
             DO 10 KK = K, MIN(K+3, 8)
              DO 10 II = I, MIN(I+3, 8)
10              A(II,J)=A(II,J)+B(II,KK)*C(KK,J)
```

Fig. 3. Example of tiling for a uniprocessor.

an innermost loop, the performance is greatly degraded. This is because, normally, many communications are required for the same array element which occurs in an innermost loop. In Fig. 3, when communication is performed just before the tiled loop (DO 10 $KK = \cdots$), communication for accessing the same region of array B is required more than once. This is inefficient. The key to increasing the performance is to vectorize communication for nonlocal data accesses in outermost loops as much as possible. When tiling is applied to the nested loop, we must avoid generating redundant communications for accessing the same array elements by reuse.

4.2.2. Group-Temporal Reuse

We discuss references that have the same affine subscript expressions $f_1(\mathbf{i}) = H\mathbf{i} + \mathbf{a}_1$ and $f_2(\mathbf{i}) = H\mathbf{i} + \mathbf{a}_2$. Group-temporal reuse occurs between two such references with distance $\mathbf{r}$ if there are iterations $\mathbf{i}_1$ and $\mathbf{i}_2$ such that $H\mathbf{i}_1 + \mathbf{a}_1 = H\mathbf{i}_2 + \mathbf{a}_2$, that is, $H(\mathbf{i}_1 - \mathbf{i}_2) = H\mathbf{r} = \mathbf{a}_2 - \mathbf{a}_1$. To determine whether such an $\mathbf{r}$ exists, we solve the system of equations to obtain a particular solution $\mathbf{r}_p$. The general solution, $\ker H + \mathbf{r}_p$, is called the group-temporal reuse vector R_{GT}. On a uniprocessor, this reuse reduces the number of memory references.

In contrast, on distributed memory machines, we require that the compiler supports message coalescing in order to exploit group-temporal reuse for nonlocal data accesses. This allows multiple operands to access the same array elements in order to refer to the same message buffer. Therefore, group-temporal reuse information helps reduce the number of communications.

4.3. Usage of Reuse Information

Our algorithm applies loop interchange to move the loop indices to positions further out than those in which reuse occurs, because this ensures that the compiler does not generate redundant communication for the same array elements that are reused in the innermost loop. It then applies tiling to the loop indices in which reuse does not occur in order to overlap communication with computation, tiling only the loop indices that require communication. We therefore apply tiling according to Condition 1.

Condition 1. Tiling is applied to loop indices that require communication and in which neither self-temporal nor group-temporal reuse occurs. These are the indices in which either the self-temporal reuse vector or the group-temporal reuse vector has a value of zero.

5. ALGORITHM FOR LOOP TRANSFORMATIONS

In this section, we present an algorithm for tiling and loop interchange in order to overlap communication with computation using a framework that combines information on data dependence, communication, and reuse. The algorithm is structured in the following five steps:

1. The In Set(2) is calculated as a set of array index vectors that require interprocessor communication (by nonlocal data accesses).
2. The communication vector(28) is calculated as a vector that indicates, on the basis of the loop index, if interprocessor communication is required.
3. A check is carried out to determine whether interprocessor communication can be vectorized.
4. The loop indices for tiling are determined by using the reuse vector and communication vector.
5. Loops are interchanged in order to move these loops indices to outer nested loops, and tiling is done to generate loops which allow overlapped communication and computation.

Figure 4 shows the algorithm for determining the loop indices to which tiling is applied (steps 1–4). In the result of the algorithm, let the tiling vector **t** be a vector showing the loop indices to which tiling is to be applied. Tiling can be applied to the indices corresponding to nonzero elements of **t**. The variable *top* shows that the outermost loop index can be a target for loop interchange.

We now explain our algorithm, in which loop interchange and tiling are applied in order to overlap communication with computation, using the program in Fig. 2 as an example.

Algorithm

```
IN  : (  $\vec{l}$ : loop index vectors, $\vec{u}$ : loop index vectors,          /* lower and upper bound */
         n: integer /* depth of a nested loop */,  l: left hand side operand,
         R: set of right-hand side operands, F: set of access functions for operands,
         G: set of distribution functions for operands,
         P: set of processor index vectors, D: set of dependence vectors)
OUT : (  $\vec{t}$ : index vector /* tiling vector */, top: integer)
i, m, top, inner: integer;
$\vec{i}$, $\vec{d}$ : loop index vector;
$\vec{y}$, $\vec{z}$ : array index vector;
$Q(P)$ : set of loop index vectors;
$r$ : right-hand side operand;
$Y_r(P)$ : set of array index vectors for operand;
$\vec{b}_r$ : loop index vector for operand $r$;
$f_r(\,)$ : access function for operand $r$;
$H_r$ : mapping matrix for operand $r$;
$g_r(\,)$ : distribution function for operand $r$;

/* step 1: calculate In Set */
foreach $\vec{p} \in P$ do
    $Q(\vec{p}) := \varnothing$;
    foreach $\vec{i} \in [\vec{l} : \vec{u}]$ do
        if $g_l(f_l(\vec{i})) = \vec{p}$ then $Q(\vec{p}) := Q(\vec{p}) \cup \vec{i}$;
    end foreach
end foreach

foreach $r \in R$ do
    foreach $\vec{p} \in P$ do
        $Y_r(\vec{p}) := \varnothing$;
        foreach $\vec{i} \in Q(\vec{p})$ do
            if $g_r(f_r(\vec{i})) \neq \vec{p}$ then
                $Y_r(\vec{p}) := Y_r(\vec{p}) \cup f_r(\vec{i})$;
            endif
        end foreach
    end foreach
end foreach
if $\left(\bigcup_{r \in R} \bigcup_{\vec{p} \in P} Y_r(\vec{p})\right) = \varnothing$ then return ( o , 0);

/* step 2: generate Communication Vector */
foreach $r \in R$ do
    $\vec{z}$ := o ;
    m = rank of operand r;
    for i := 1 to m do
        if $\left(\bigcap_{\vec{p} \in P} \{y_i \mid \vec{y} = (y_1, \ldots, y_m), \vec{y} \in Y_r(\vec{p})\}\right) = \varnothing$ then
            $\vec{z} = \vec{z} \cup span\{\vec{e}_i\}$
        endif
    end for
    $\vec{b}_r := H_r^T \vec{z}$;
    if $\vec{b}_r \neq$ o then
        for i := 1 to n do
            if $\exists span\{\vec{e}_i\} \in \ker f_r$ then
                $\vec{b}_r = \vec{b}_r \cup span\{\vec{e}_i\}$;
            endif
        end foreach
    endif
    /* step 3: check condition 2 */
    if $\exists \vec{d} \in D : (\vec{d}$ is anti dependence by oprerand $r$ and $\vec{d} \cap \vec{b}_r \neq \varnothing)$ and
    $\exists \vec{d} \in D : (\vec{d}$ is true dependence by operand $r$ and $\vec{d} \cap \vec{b}_r \neq \varnothing)$ then
        return ( o , 0);
    end if
end foreach

/* step 4: calculate $\vec{t}$ */
$\vec{t}$ = o ;
foreach $r \in R$ do
    for i := 1 to n do
        if ( $(\ker f_r \cap span\{\vec{e}_i\} = \varnothing)$ or $((\ker f_r + \vec{r}_p) \cap span\{\vec{e}_i\} = \varnothing)$ )
            and $\vec{b}_r \cap span\{\vec{e}_i\} \neq \varnothing$ then
                $\vec{t} := \vec{t} \cup span\{\vec{e}_i\}$;
        end if
    end foreach
end foreach
if $\vec{t}$ = o then return ( o , 0);

/* determine fully permutable nest from top to n */
top := 1;
if $D = \varnothing$ then return ( $\vec{t}$ , top);
for inner := n to 1 by -1 do
    if $t_{inner} \neq 0$ then break;
end for

while (top < inner) do
    if $\forall \vec{d} \in D : ((d_1, \ldots, d_{top-1}) \succ 0$ or $\forall top \leq i \leq n : d_i \geq 0)$ then
        break;
    top := top + 1;
end while
if (top = inner) return ( o , 0);
for i := 1 to top-1 do $t_i$ := 0;
return ( $\vec{t}$ , top);
```

Fig. 4. Algorithm for determining the loop indices to which tiling is applied (steps 1–4).

Example 1. The input parameters to the algorithm in Fig. 4 are as follows:

$$n = 3, \quad \mathbf{l} = (1, 1, 1), \quad \mathbf{u} = (8, 8, 8), \quad P = \{(0), (1)\},$$

$$l = \{A(I, J)\}, \quad R = \{A(I, J), B(I, K), C(K, J)\}, \quad D = \varnothing$$

5.1. Calculating the In Set (Step 1)

First, for a loop to be executed in parallel, its iteration space is partitioned and each iteration sub-space is allocated to a single processor. In this paper, we use the owner-computes rule,[13, 29] but this is not a restriction imposed by our algorithm. The left-hand side of each assignment statement in a loop is used to calculate the iteration space, which enables a processor to store local data. We define the Local Iteration Set $Q(\mathbf{p})$ as the set of loop index vectors that access array elements on the processor $\mathbf{p}$.

The compiler then calculates the set of array references that causes nonlocal data access when each iteration of the partitioned iteration space is executed. The accessed elements of the right-hand side array r are determined by the Local Iteration Set $Q(\mathbf{p})$ and the subscript expression $f_r(\mathbf{i})$ of the array r. We define the In Set $Y(\mathbf{p})$ as the set of array index vectors that causes nonlocal data accesses when each loop index of the Local Iteration Set $Q(\mathbf{p})$ is executed on the processor $\mathbf{p}$.

Example 2. In Fig. 2, the subscripts of the left-hand side array A are (I, J). The array A is distributed in $(*, \text{BLOCK}(4))$. We assume $l = \{A(I, J)\}$ to be the left-hand side operand. The mapping functions f_l and the distribution function g_l for the operand $A(I, J)$ and the Local Iteration Set $Q(\mathbf{p})$ are as follows:

$$f_l(\mathbf{i}) = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \mathbf{i}, \quad g_l(\mathbf{j}) = \left(\left\lfloor \frac{j_2 - 1}{4} \right\rfloor \right),$$

$$Q((0)) = (1:4, 1:8, 1:8), \quad Q((1)) = (5:8, 1:8, 1:8)$$

In Sets are calculated for all right-hand side operands in the loop. For this example, the results of the operands without communication are omitted, because they do not affect the results of loop transformations. We show the result in array B that need to be communicated. In Fig. 4, we assume $r = R = \{B(I, K)\}$ to be the right-hand side operand. The subscripts of the right-hand side array B are (I, K). The array B is distributed in $(*, \text{BLOCK}(4))$. The mapping functions f_r and distribution function g_r for the operand $B(I, K)$ and the In Set Y_r are as follows:

$$f_r(\mathbf{i}) = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \mathbf{i}, \quad g_r(\mathbf{j}) = \left(\left\lfloor \frac{j_2 - 1}{4} \right\rfloor \right),$$

$$Y_r((0)) = (1:8, 5:8), \quad Y_r((1)) = (1:8, 1:4)$$

5.2. Generating the Communication Vector (Step 2)

We now present a method of generating the communication vector, which represents communication information, based on the loop index. We want to calculate communication information based on the array index, the data dependence vector, and the reuse vector based on the loop index. H_r in $f_r(\mathbf{i})$ maps the loop index vector to the array index vector.

If the depth n of the nested loop is smaller than the rank m of the array r, the communication information is reduced when it is mapped to the loop index. This occurs because the matrix does not completely project an array index vector into the loop index vector. Practically speaking, if the ith value of $\ker H_r$ is nonzero, it shows that the ith loop index variable does not appear in the subscripts. If the ith index variable for the nested loop does not appear in the subscripts of the right-hand side array, the compiler has to add the information to the ith index conservatively in order to show that communication occurs. We define the result vector, called the communication vector, as follows. The communication vector $\mathbf{b} = (b_1, \ldots, b_n)$, $b_i =$ nonzero or zero, is a vector in which each elemental value represents whether the processors require communication, on the basis of each loop index.

Example 3. In Fig. 4, the range of the first dimension of the In Set for array B is the same on each processor. Each processor reads a different range of the second dimension of the In Set. As a result, the In Set shows that communication is required in the second dimension of array B. The compiler then calculates the loop index for accessing the second dimension of array B. In Fig. 4, it is calculated from the following expression:

$$\mathbf{b}_r = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}^T \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

The result is $\mathbf{b}_r = \operatorname{span}\{\mathbf{e}_3\}$. We use the vectors $\mathbf{e}_1$, $\mathbf{e}_2$, and $\mathbf{e}_3$ to represent (1, 0, 0), (0, 1, 0) and (0, 0, 1), respectively. The result shows that the loop index variable K accesses the second dimension of array B.

To the loop index whose variable J does not appear in the subscript $B(I, K)$, the compiler adds the vector $\operatorname{span}\{\mathbf{e}_1\}$ that needs to be communicated. Here, the communication vector is $\mathbf{b}_r = \operatorname{span}\{\mathbf{e}_1, \mathbf{e}_3\}$.

5.3. Determining if Communication can be Vectorized (Step 3)

To determine an appropriate mode of communication, we define the communication dependence vector $\mathbf{c}$ as a vector in which the value of each element represents whether communication occurs as a result of anti dependence $\mathbf{d}_a$ or true dependence $\mathbf{d}_t$ for the operand. It consists of an anti communication dependence vector $\mathbf{c}_a$ and a true communication dependence vector $\mathbf{c}_t$. These are then derived, as follows, by combining information on data dependence and communication:

$$\mathbf{c}_t = (c_{t1}, c_{t2}, \ldots, c_{tn}), \qquad c_{ti} = \begin{Bmatrix} d_{ti}\ (\text{if } b_i \neq 0) \\ 0\ (\text{if } b_i = 0) \end{Bmatrix},$$

$$\mathbf{c}_a = (c_{a1}, c_{a2}, \ldots, c_{an}), \qquad c_{ai} = \begin{Bmatrix} d_{ai}\ (\text{if } b_i \neq 0) \\ 0\ (\text{if } b_i = 0) \end{Bmatrix}$$

The compiler can apply message vectorization if Condition 2 is satisfied.(28)

Condition 2. Message vectorization is applied if an operand satisfies exactly one of the following conditions:

1. It has only an anti communication dependence vector $\mathbf{c}_a$.
2. It has only a true communication dependence vector $\mathbf{c}_t$.
3. It has no data dependence.

If the operand has only a true communication dependence vector $\mathbf{c}_t$, it requires vector pipeline communication. Otherwise, the operand requires vector prefetch communication.

Example 4. $D = \emptyset$ shows that any operand in the loop has no dependence, and thus it satisfies Condition 2. The communication for array B can be vectorized by using prefetch communication.

5.4. Determining Appropriate Loop Indices for Tiling (Step 4)

To determine loop indices that satisfy Condition 1, we provide a framework for combining information on reuse and communication as follows. The compiler calculates the bitwise **AND** of the communication vector **b** and the logical negative of the self-temporal reuse vector and the group-temporal reuse vector. The ith zero value of the reuse vector shows that reuse does not occur in the ith loop index. The ith non-zero value of the communication vector shows that communication occurs in the ith loop index. If the ith value of the reuse vector is zero and the ith value of the communication vector is nonzero, the compiler sets the ith nonzero value of the result vector. Then, if the ith value of the result vector is nonzero, the compiler determines that tiling can be applied to the ith loop index. It determines the scope of the tiled loop by examining whether the nested loops are fully permutable.[19]

Example 5. In the example, we focus only on self-temporal reuse, because not more than one reference is made to the same array with communication. Thus, we assume that there is no group-temporal reuse. In Fig. 2, the self-temporal reuse vector R_{ST} for the operand $B(I, K)$ is $\mathrm{span}\{\mathbf{e}_1\}$. The bitwise **AND** of $\mathbf{b}_r = \mathrm{span}\{\mathbf{e}_1, \mathbf{e}_3\}$ and $\mathrm{span}\{\mathbf{e}_2, \mathbf{e}_3\}$ is $\mathbf{t} = \mathrm{span}\{\mathbf{e}_3\}$.

To ensure legality of the loop transformations, the compiler then determines the scope of the nested loop that is fully permutable. Since $D = \emptyset$ in the loop, the whole nested loop is fully permutable. The range of the loop index is between ($top =$) 1 and ($inner =$) 3. As a result, the algorithm returns $\mathbf{t} = \mathrm{span}\{\mathbf{e}_3\}$ and $top = 1$ in Fig. 4.

5.5. Apply Loop Interchange and Tiling (Step 5)

First, the compiler determines the loop index to which tiling is to be applied on the basis of the communication method in the loop. If the operands in the loop require only vector prefetch communication, tiling is applied to all loop indices. This is because the loop can be executed without synchronization after communication is completed, and thus the parallelism is not varied when tiling is applied to any loop index. If the operands in the loop also require vector pipeline communication, tiling is applied to the loop index that accesses the array dimension distributed among the fewest processors, because the array size for each communication should be as small as possible to increase the effectiveness of overlap.

Next, in order to perform communication for a tiled loop at the outermost possible level, the compiler interchanges the determined loop indices with the indices of outermost loop that is fully permutable. The compiler then applies tiling to the loop indices that were moved into the outermost loop.

Example 6. The loop has only prefetch communication. Since only the third index has a nonzero value in the vector $\mathbf{t}$, the compiler interchanges the loop index variable K with the outermost loop index. Finally, it applies tiling to the new outermost loop.

6. CODE GENERATION

In this section, we describe methods for generating communication such that it is overlapped with computation. The method depends on the

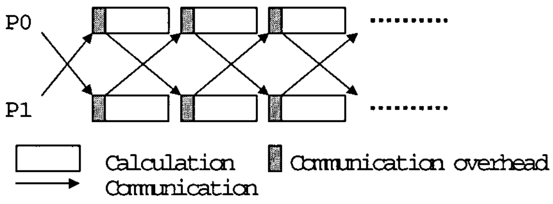


Fig. 5. Overlapping of computation with prefetch communication.

communication method of each operand, which may be prefetch or pipeline communication. We assume that a nonblocking communication interface is implemented on distributed memory machines. Non-blocking communication means that the next computation can be executed without waiting for a reply after a command to send or receive data has been issued. In this paper, SEND/RECEIVE indicates a function that issues a non-blocking send/nonblocking receive. WAIT indicates a function that waits for all data to be received.

6.1. Overlap for Prefetch Communication

An operand that requires prefetch communication must reference elements before they are written. To overlap communication with computation of the tiled loop, the processor performs prefetch communication that exchanges the two receive buffers alternatively. First, the processor sends array elements that will be read in the next tiled loop to another buffer before the execution of the tiled loop. Then, it starts to calculate the tiled loop, using the current buffer. Here, the processor can overlap communication with computation. Before starting to calculate the next tile, the processor waits for the arrival of the data sent to the buffer, in order to synchronize itself with the other processor. After exchanging the old buffer with the received buffer, it starts to calculate the tiled loop. Figure 5 shows

```
   DO 10 K=1,8,KB
    if (K=1) SEND & RECEIVE B' for the first tile
    WAIT
    swap B' and B
    if (K<=8-KB) SEND & RECEIVE B' for the next tile
    DO 10 KK=K,MIN(K+KB-1,8)
     DO 10 J=1,8/2
      DO 10 I=1,8
10     A(I,J)=A(I,J)+B(I,KK)*C(KK,J)
```

Fig. 6. Example of prefetch communication.

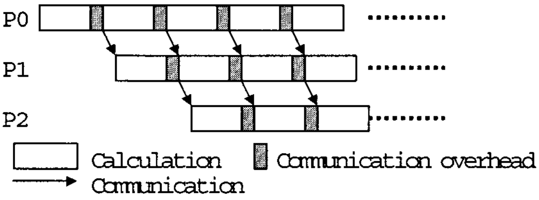


Fig. 7. Overlapping of computation with pipeline communication.

how communication and computation are performed. Figure 6 shows the pseudo-code generated for the program in Fig. 2.

6.2. Overlap for Pipeline Communication

The operand that requires the pipeline communication to be written at iteration **i** is read at iteration **i** + **d**. The processor sends the array elements to another processor that references them after the calculation of the tiled loop is completed, and starts to calculate the next tiled loop during the communication. Thus, the processor can overlap communication with computation. The other processor then executes the next tiled loop after receiving data. Figure 7 shows how communication and computation are performed. Figure 8 shows the original program and the generated pseudo-code.

7. EXPERIMENTS

In this section, we give the experimental results obtained by using our algorithm, which we have implemented in our HPF compiler.[28, 30] To measure the effectiveness of the algorithm, we compiled two applications written in HPF. One is a matrix multiplication code that requires prefetch communication. The other is a Successive Over Relaxation (SOR) code that requires prefetch and pipeline communication.

```
*HPF$ DISTRIBUTE A(BLOCK,*)

  DO 10 J=1,N
    DO 10 I=1,M
10    A(I,J)=A(I,J-1)
```

a) Original code

```
 DO 11 J=1,N,JB
   receive A
   wait
   DO 10 JJ=J,MIN(J+JB-1,N)
     DO 10 I=1,M/n_procs
10       A(I,JJ)=A(I,JJ-1)
11   send A
```

b) Generated pseudo-code

Fig. 8. Example of pipeline communication.

For our experiments, we used an IBM SP with 32 thin nodes. The high-performance switch (HPS) provides a DMA mechanism between the network and the system buffer. We used the IBM Message Passing Library,(31) which provides a nonblocking communication interface.

7.1. Matrix Multiplication

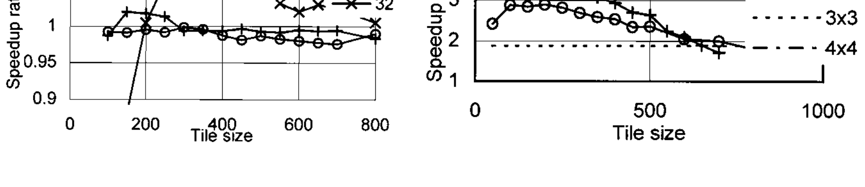
We ran the matrix multiplication code with 1600×1600 arrays distributed in (*, BLOCK). The performance results are shown in Fig. 9a. The vertical axis shows the speedup ratio normalized by the execution time without communication overlap, while the horizontal axis shows the size of tile used by the compiler. In this experiment, a tile size between 200 and 500 was found to be the best choice.

7.2. Successive Over Relaxation Method

For the second experiment, we ran the SOR code with 2000×2000 arrays distributed in (BLOCK, *). The performance results are shown in Fig. 9b. The vertical axis shows the speedup ratio normalized by the execution time of the sequential code on a single processor, while the horizontal axis shows the tile size used by the compiler. The results of running SOR code without our optimization are also shown for 3×3 and 4×4 processor configurations of in the array distribution (BLOCK, BLOCK). In this experiment, a tile size between 100 and 200 was found to be the best choice.

7.3. Discussion

Prefetch communication is generated in the matrix multiplication code. As the number of processors increases, the effectiveness of the algorithm becomes apparent for a wider range of tile sizes. The performance gains also increase when more processors are used. In the SOR code, both



(a) Matrix multiplication code (b) SOR code

Fig. 9. Performance results of the experiments.

prefetch communication and pipeline communication are generated. The program is executed in a wavefront method. The performance with our algorithm is better than the performance when the array distribution is done by the user.

The best tile size is determined by the relationship between the communication time among processors and the execution time of the tiled loop. When there is prefetch communication, identically, if the communication is issued before executing the nested loop, the communication time is hidden. Since our algorithm does not move prefetch communication out of the nested loop, communication takes up less of the execution time if the communication time is hidden by the calculation time of the tiled loop. The best tile size gives a ratio of 1:1 for the communication time and the calculation time of the tiled loop. The performance drops when the tile size is larger than optimum in the case of matrix multiplication by 32 processors. When there is pipeline communication, the parallelism increases and the amount of data in each communication decreases if the tile size is smaller, but communication overhead increases with the total number of communications. On the other hand, the parallelism decreases if the tile size is larger. Actually, in Fig. 9, the performance drops when the tile size is 50 and larger than 200 by 32 processors. In an SP thin system, we can overlap computation with transfer over a network, but we cannot overlap computation with the transfer between the system buffer and the user buffer. This is because a DMA transfer in an SP system only supports copying of data between the network and the system buffer. In an SP system, the following two types of overhead exist when communication and computation are overlapped by using a non-blocking communication interface:(31)

1. An interrupt to the processor, generated by the DMA mechanism at the completion of the data transfer from the network to the system buffer.
2. A copy between the system buffer and the user buffer by the processor.

Roughly speaking, the first type of overhead takes $60\,\mu$s, and the second type of overhead takes $28\,\mu$s/KB. These overheads also bring unnecessary data (for computation) into the data cache, and thus decrease the data locality.

Despite the overhead of an SP mechanism, our algorithm improves the performance by choosing an appropriate tile size. If a system supports direct data transfer between the network and the user buffer on the processor, we would be able to obtain an even better speedup.

8. CONCLUSIONS

We have described an automatic loop transformation algorithm that hides the communication latency by overlapping communication with computation. Previously, this type of optimization could only be achieved by hand-coding programs. We developed the algorithm by combining the vectors of data dependence, communication, and reuse. We have also described how to generate communication for tiled loops. We implemented this algorithm in our HPF compiler, and experimental results have shown its effectiveness. We are currently working on a heuristic algorithm to determine an appropriate tile size for overlapping communication with computation.

ACKNOWLEDGMENTS

We are grateful to Osamu Gohda, Gyo Ohsawa, Toshio Suganuma, and Takeshi Ogasawara of Tokyo Research Laboratory for implementing our HPF compiler and for participating in helpful discussions.

REFERENCES

1. Stanford SUIF Compiler Group, SUIF: A Parallelizing and Optimizing Research Compiler, Technical Report, Stanford University, CSL-TR-94-620 (1994).
2. C. W. Tseng, An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines, Ph.D. thesis, Rice University, CRPC-TR93291 (1993).
3. H. P. Zima, H. J. Bast, and M. Gerndt, SUPERB: A Tool for Semiautomatic MIMD/SIMD Parallelization, *Parallel Computing* **6**:1–18 (1988).
4. Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka, Fortran90D/HPF Compiler for Distributed Memory MIMD Computers: Design, Implementation and Performance Results, *Proc. Supercomputing*, pp. 351–360 (1993).
5. P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, and S. Ramaswamy, The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers, *Proc. First Int'l Workshop on Parallel Processing*, pp. 322–330 (1994).
6. T. Shindo, H. Iwashita, T. Doi, J. Hagiwara, and S. Kaneshiro, HPF Compiler for the AP1000, *Proc. Int'l. Conf. Supercomputing*, pp. 190–194 (1995).
7. High Performance Fortran Forum, High Performance Fortran Language Specification, Version 1.0, Technical Report, Rice University, CRPC-TR92225 (1992).
8. S. Hiranandani, K. Kennedy, and C. W. Tseng, Compiling Fortran D for MIMD Distributed-Memory Machines, *Comm. ACM* **35**:66–80 (1992).
9. D. E. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. Eicken, and K. Yelick, Parallel Programming in Split-C, *Proc. Supercomputing*, pp. 262–273 (1993).
10. A. Lain and P. Banerjee, Techniques to Overlap Computation and Communication in Irregular Iterative Applications, *Proc. Int'l Conf. Supercomputing*, pp. 236–245 (1994).
11. S. Hiranandani, K. Kennedy, and C. W. Tseng, Preliminary Experiences with the Fortan D Compiler, *Proc. Supercomputing*, pp. 338–350 (1993).

12. T. Horie, K. Hayashi, T. Shimizu, and H. Ishihata, Improving AP1000 Parallel Computer Performance with Message Communication, *20th Ann. Int'l Symp. Computer Architecture*, pp. 314–325 (1993).
13. A. Rogar and K. Pingali, Process Decomposition Through Locality of Reference, *Proc. SIGPLAN '89 Conf. Progr. Language Design and Implementation* (1989).
14. D. J. Palermo, E. Su, J. A. Chandy, and P. Banerjee, Communication Optimizations Used in the PARADIGM Compiler for Distributed-Memory Multicomputers, *Proc. 23rd Int'l Conf. Parallel Processing*, pp. II:1–10 (1994).
15. H. Ohta, Y. Saito, M. Kainaga, and H. Ono, Optimal Tile Size Adjustment in Compiling General DOACROSS Loop Nests, *Proc. Int'l Conf. Supercomputing*, pp. 270–279 (1995).
16. U. Banerjee, Unimodular Transformations of Double Loops, *Proc. Workshop on Advances Lang. Compilers for Parallel Processing*, pp. 192–219 (1990).
17. M. Wolfe, *High Performance Compiler for Parallel Computing*, Addison-Wesley Publishing Company, (1995).
18. M. Wolfe, More Iteration Space Tiling, *Proc. Supercomputing*, pp. 655–664 (1989).
19. M. E. Wolfe and M. S. Lam, A Loop Transformation and Theory and an Algorithm to Maximize Parallelism, *IEEE Trans. Parallel Distrib. Syst.* **2**(4):452–471 (1991).
20. T. Agewara, J. L. Martin, J. H. Mirza, D. C. Sadler, D. M. Dias, and M. Snir, SP2 System Architecture, *IBM Syst. J.* **344**(2):152–184 (1995).
21. M. E. Wolfe and M. S. Lam, A Data Locality Optimizing Algorithm, *Proc. ACM SIGPLAN Conf. Progr. Lang. Design and Implementation*, pp. 30–44 (1991).
22. C. Koelbel, P. Mehrotra, and J. V. Rosendale, Supporting Shared Data Structures on Distributed Memory Architectures, *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pp. 177–186 (1990).
23. R. Hanxlenden and K. Kennedy, GIVE-N-TAKE: A Balanced Code Placement Framework, *Proc. ACM SIGPLAN '94 Conf. Progr. Lang. Design and Implementation*, pp. 107–120 (1994).
24. K. Fujiwara, K. Shiratori, M. Suzuki and H. Kasahara, Multiprocessor Scheduling Algorithms Considering Data-Preloading and Poststoring, *Trans. IEICE*, D-1 **75**(8):495–503 (1992).
25. A. W. Lim and M. S. Lam, Maximizing Parallelism and Minimizing Synchronization with Affine Transforms, *Conf. Record of the 24th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Progr. Lang.* (1997).
26. J. M. Anderson, S. P. Amarasinghe, and M. S. Lam, Data and Computation Transformations for Multiprocessors, *Proc. Fifth ACM SIGPLAN Symp. on Principles and Practice of Parallel Processing* (1995).
27. Michael Philippsen, Automatic Alignment of Array Data and Processes to Reduce Communication Time on DMPPs, *Proc. Fifth ACM SIGPLAN Symp. on Principles and Practice of Parallel Processing* (1995).
28. K. Ishizaki and H. Komatsu, A Loop Parallelization Algorithm for HPF Compilers, *Eigth Workshop on Language and Compilers for Parallel Computing*, pp. 12.1–15 (1995).
29. D. Calllahan and K. Kennedy, Compiling Programs for Distributed-Memory Multiprocessors, *J. Supercomputing* **2**:151–169 (1988).
30 T. Suganuma, H. Komatsu, and T. Nakatani, Detection and Global Optimization of Reduction Operations for Distributed Parallel Machines, *Proc. Int'l Conf. Supercomputing* (1996).
32. M. Snir, P. Hochschild, D. D. Fryer, and K. J. Gildea, The Communication Software and Parallel Environment of the IBM SP2, *IBM Syst. J.* **34**(2):205–221 (1995).