6 Related Work

Many approaches have been proposed to optimize the communication in SPMD execution. These include: *Code reordering, Loop interchange, Loop reversal, Loop elimination* and *Communication combining* [1] [6] [4] [10]. The purpose of these approaches is to extract as much parallelism as possible. Instead of inserting the communication at the earliest possible point, these approaches usually insert communication immediately before the computation. Moreover, all of these approaches assume that the *send* and *receive* instructions are inserted at the same point in the program. This does not allow effective overlapping of communication and computation. Gallivan and Jalby [3] were one of the first to formally treat the problem of optimizing data transfers in distributed-memory systems by **avoiding redundant communication** and **avoiding sequentialization**. In [11], the authors defined some array reference patterns and use pre-implemented routines to match those patterns and achieve communication optimization. However, only restricted classes of communication patterns can be optimized in this way. In [9], the authors proposed optimizations similar to the ones presented in this paper. Our unified framework, however, allows us to perform all the optimizations together and enables us to decide the tradeoff between competing optimizations.

Data flow analysis has been successfully applied to allow code optimizations involving scalar variables. Only recently has the technique been applied to array variables. In [2], the author proposed a data flow analysis algorithm to collect information about array and scalar references and suggested that this information could be used for program verification and parallel program construction. We use data flow analysis for communication optimizations for which we need to collect the communication pattern information.

7 Summary

The purpose of the optimizations is to reduce communication and to increase parallelism. The contribution of this paper is twofold. First, we exploit the idea of separating the *send* and *receive* instruction to overlap communication with execution. Second, we present the data flow analysis technique for solving data communication optimization problems. We are currently investigating how to expand the framework to include interprocedural analysis.

References

[1] D. Callahan, and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," *The Journal of Supercomputing*, Vol. 2, 1988.

[2] P. Feautrier, "Dataflow Analysis of Array and Scalar References," *International Journal of Parallel Programming*, Vol. 20, January, 1991.

[3] K. Gallivan, and W. Jalby, "On the Problem of Optimizing Data Transfers for Complex Memory System," *Proc. ACM International Conference on Supercomputing*, 1988.

[4] H. M. Gerndt, and H. P. Zima, "Optimizing Communication in SUPERB," *CONPAR 90*, 1990.

[5] G. Goff, K. Kennedy, and C. Tseng, "Practical Dependence Testing," *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991.

[6] R. Gupta, "Compiler Optimizations for Distributed Memory Programs," *Proc. of the Scalable High Performance Computing Conference*, 1992.

[7] P. Havlak, and K. Kennedy, "An Implementation of Interprocedural Bounded Regular Section Analysis," *IEEE Trans. Parallel and Distributed Sys.*, Vol. 2, July, 1991.

[8] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. W. Tseng, "An Overview of the Fortran D Programming System," *Proc. of the Fourth Workshop on Languages and Compilers for Parallel Computing*, 1991.

[9] S. Hiranandani, K. Kennedy, and C. W. Tseng, "Compiling Fortran D for MIMD Distributed-Memory Machines," *Communication of ACM*, Vol. 35, August, 1992.

[10] C. Koelbel, P. Mehrotra, and J. V. Rosendale, "Supporting Shared Data Structure on Distributed Memory Architectures," *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 1990.

[11] J. K. Li, and M. Chen, "Compiling Communication-Efficient Programs for Massively Parallel Machines," *IEEE trans. on Parallel and Distributed Sys.*, Vol. 2, July, 1991.

[12] D. E. Maydan, J. L. Hennessy, and M. S. Lam, "Efficient and Exact Data Dependence Analysis," *ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991.

[13] Thinking Machines Corporation, "CM Fortran Reference Manual," 1991.

[14] W. Pugh, and D. Wonnacott, "Eliminating False Data Dependences Using the Omega Test," *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation*, 1992.

[15] M. J. Quinn, and P. J. Hatcher, "Compiling SIMD Programs for MIMD Architectures," *Proceedings of International Conference on Computer Languages*, 1990.

[16] "Fortran 90 Standard," 1991.

Algorithm 4: Optimization.
Input: A program P with array reference information U at each point;
Output: The optimized program.

Optimize(P, U)
  For each statement S of P
    check the two array references sets before and after S: $in(S)$, $out(S)$
    If there is $u \in out(S)$ but $u \notin in(S)$ then
      let the point immediately after S be S^u;
  For each statement S
    For each array reference u that is referenced in S
      find the point S_u by backward searching;
  For each array reference u
    If S^u is earlier than S_u then
      choose between S^u and S_u the best point C_u at which u can be combined into the largest the largest message;
    else choose S^u as C_u;
  For each array reference u Do
    insert a parallel loop at C_u send elements of u.

Figure 10: Optimization Algorithm

5 Extensions of the Basic Data Flow Analysis Algorithm

Suppose the source function is a linear function of the form: $Source(i) = ai + c$ with $a \neq 1$. In this situation, the range of an array reference is no longer a set of consecutive integers, thus complicating the implementation of the *splitting* and *combining* operations. The implementations of these operations are even harder for arbitrary source and destination functions. In any case, we must adopt a conservative approach. Being conservative has the following implications. First, for *splitting* operation, if we do not know exactly which portion of u is redefined by v, we should guarantee that no element of u that is actually redefined by v will be propagated. In this situation, existing data flow dependence tests [7] [14] [12] [5] can be used to determine whether two array references are independent or not. Based on the result, either the entire array reference is propagated or nothing is propagated. Second, for *combining* operation, if we are not sure that two array references u and v have the same communication pattern, we simply do not combine the array references. This impacts the performance of the data flow algorithm not its precision.

So far, we have considered only unidimensional arrays. For multi-dimensional arrays in nested loops, the situation becomes more complicated. In some situations, however, the loop can be collapsed and the array can be linearized. In these cases, the techniques of Section 4 can be applied. Obviously, this approach is not applicable to all loops. Thus, extensions are required to handle multidimensional arrays and multiply nested loops. For two dimensional arrays within doubly nested loops, the *Source* function produces a vector of 2 values. If the first value of $Source(i, j)$ is a shift function of i and the second value of $Source(i, j)$ is a shift function of j, then we can do *splitting* and *combining* operations on the two dimensions independently. Consider the following loops:

```
Do i=1, 60
  Do j=1,100
  S1: C[i,j] =..B[i,j]..
Do i=50, 100
  Do j=1, 100
  S2: A[i, j] = ...B[i, j]*C[i+1, j+2]
```

In this example, we can propagate the following two array references at point S_2 above the second loop:

$$u =< B, (i = 50, 100; j = 1, 100), (i; j) >,$$
$$v =< C, (i = 50, 100; j = 1, 100), (i + 1; j + 2) >$$

and u will be combined with the array reference at point S_1 to become $u' =< B, (i = 1, 100; j = 1, 100), (i; j) >$ and v will be split into $v' =< C, (i = 60, 100; j = 1, 100), (i; j) >$. We can do the same for n dimensional arrays within n nested loops.

Now let's consider a different situation involving arrays defined and referenced at different level of a nested loop.

```
S1: Do i=1, 100
S2:   Do j=1, 100
S3:     B[i,j]= Exp.
S4:     Do k=1, 100
S5:       A[j,i] = ... B[i,k]...
```

In this nested loop, we have an array reference u and an array definition v: The problem here is that at point S_3 only one element of u is redefined by v and at point S_1 all elements of u are redefined. For this case, the best communication scheme would be to send the whole B before the point S_1 and then send $B[i, j]$ again after point S_3. Since $B[i, j]$ is sent twice, we need to receive it twice to get the right value and the first *receive* can be performed immediately after the second *send*:

```
Do i=1, 100
  Do j=1, 100
    Do k=1, 100
    S(B[i,k], P_{i,k} → P_{j,i})
Do i=1, 100
  Do j=1, 100
    B[i,j]= Exp.
    S(B[i,j], P_{i,j} → P_{j,i})
    R(B[i,j],P_{j,i} ← P_{i,j})
    Do k=1, 100
      R(B[i,k],P_{j,i} ← P_{i,k})
      A[j,i] = ... B[i,k]...
```

We need to develop some test conditions under which we can introduce limited amount of extra communication.

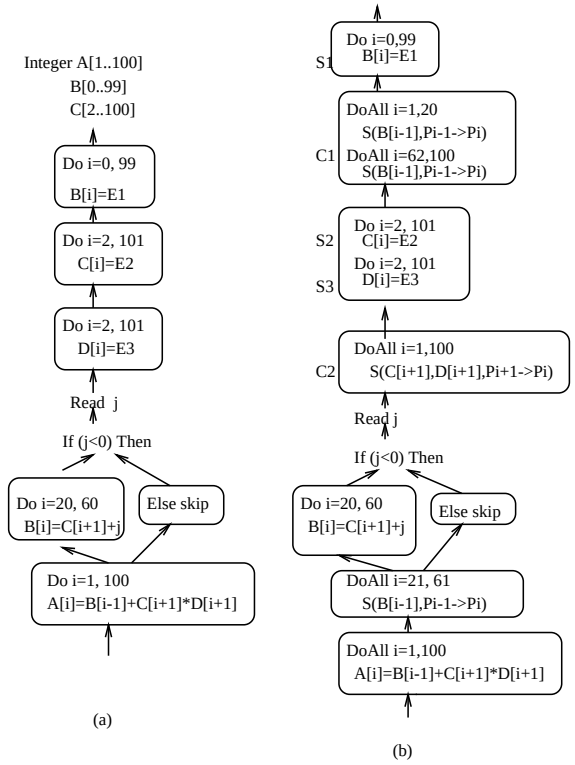

Figure 9: (a) The program with data flow information; (b) After optimization.

4. By (11) and (12):$ref(S_5)$={<C,(i=20,60),i+1>}; $def(S_5)$={<B,(i=20,60),i>}; and by (13): $in(S_5)$={<D,(i=1,100),i+1>, Split(<B,(i=1,100),i-1>,<B,(i=20,60),i>), Widen(<C,(i=1,100),i+1>,<C,(i=20,60),i+1>)}

   ={<B, (i=1, 20), i-1>,<B, (i=60, 100), i-1>, <C,(i=1,100),i+1>,<D,(i=1,100),i+1>}

5. $in(S_6) = out(S_6)$;

6. By (10): $in(S_4) = in(S_5)$;

7. The rest of the computation is straightforward: $in(S_3)$={<C, (i=1, 100), i+1>, <B, (i=1, 20), i-1>, <B,(i=60+2,100),i-1>} $in(S_2)$={<B, (i=1, 20), i-1>, <B, (i=62, 100), i-1>} $in(S_1) = \{\}$.

4 An Optimization Algorithm

In this section we develop an optimization algorithm which takes advantage of the information computed in the preceding section. In order to facilitate the implementation of the optimizations, we modify the data flow sets as follows. If an array reference is split during the computation of $in(S)$ then we modify $out(S)$ by rewriting the reference as the split portions, which are in $in(S)$, and the remaining portion. All sets of statements after S are also modified similarly. These modifications can be carried out in a single top-down pass of the syntax tree. For example, in the program of Figure 9 (a), since B is split at S_5 we rewrite the reference B in $out(S_5) = out(S_6) = in(S_7)$ as: $\{< B,(i = 1,20), i-1 >, < B,(i = 21,61), i-1 >, < B,(i = 62,100), i-1 >\}$, which results from splitting $< B,(i = 1,100), i-1 >$. Next we insert the communication primitives into the program at the earliest point at which the communication can be performed. The earliest point for a communication can be found as follows:

For each array reference u, identify the statement S such that $u \in out(S)$ but $u \notin in(S)$. This S is the earliest point for the communication u.

This algorithm achieves the optimizations of *avoiding sequentialization, overlapping* and *avoiding redundancy*. In order to achieve the *combining of smaller messages*, we need to modify the above method. One observation is that we do not need to insert a data *send* at the earliest possible point. Sometimes it is better to insert it at a later point so it can be combined with another data *send*. For example, consider the program of Figure 9 (a) again. We can combine the sending of C and D if we do not insert the sending for C at the earliest point. However, not inserting the *send* at the earliest point may affect other optimizations, such as *avoiding sequentialization* and *overlapping*. Therefore there is some interaction among the different optimizations and the degree to which various optimizations are applied must be balanced. One way of balancing these optimizations is to find, for each array reference u, a point above which the communication of u results in complete overlapping of communication with computation. For this purpose, we introduce the following definition:

Def. 7. Given a program P, for any array reference u in P, S^u is the earliest possible point where the communication required by u can be performed and S_u is the point at which or before which we can perform the communication yet allowing enough overlapping.

We already know how to compute S^u. In order to compute S_u, suppose u is referenced at point S_1 and S_2 is earlier than S_1. If between points S_2 and S_1 the amount of computation performed by the consumer processor is larger than some given threshold, T, then we can choose S_2 as S_u. The value of T depends on the communication cost in the system. It should be large enough to hide the communication latency by providing sufficient overlap of communication with computation. An outline of the optimization algorithm is given in Figure 10. Applying this algorithm on the program of Figure 9 (a), we get the program of Figure 9 (b) in which we only show where the *send* instructions are inserted. The *receive* instructions should be inserted at the points where the data are actually needed.

$$def(S) = \{< ID_1, D, f(i_1, \ldots, i_n), f(i_1, \ldots, i_n) >\} \quad (12)$$

$$in(S) = (out(S) -_{<>} def(S)) \cup_{<>} ref(S) \quad (13)$$

By the definition of $in(S)$, any array reference that is defined in S can not be in $in(S)$ and any array reference that is used in S is in $in(S)$.

Using these equations, we can collect all *in* and *out* information by traversing the syntax tree.

3.3 Solving data flow equations

In this section, we describe algorithms for solving data flow equations. The programs that we can handle include an important class of programs: *singly nested loops* and *unidimensional arrays*. Furthermore, we make the following assumptions:

1. The loop step m is always 1. This is not a crucial assumption since every loop can be converted into an equivalent form with step $m = 1$. We also assume that the loop bounds are compile time constants.

2. Both the source function and the destination function are shift functions, $Source(i) = i + c$ and $Dest(i) = i + c'$ where c, c' are integer constants. We convert the destination function into an identity function by changing the domain to $(i = l + c', h + c')$ and the $Source$ function to $Source(i) = i + c - c'$. Hence we assume that the destination function is the identity function.

We can also expand our framework to cover cases like *block* and *cyclic* data distribution. For block distribution, we can assign a new index value to the collection of elements that are in the same block; for cyclic distribution, we simply divide the whole array into several subarrays ans give each subarray a new name. Under the above assumptions, both the domain and the range of an array reference will be a sets of consecutive integers and therefore can be represented by two integers $D_u = (i = l, h)$ and $R_u = (l + c, h + c)$. In the remainder of this section, we will omit the destination function from the reference tuples since it is always assumed to be the identity function. The *Split, Widen* and *Narrow* operations for such array references are given in Figure 6, Figure 7 and Figure 8.

We apply the data flow analysis algorithm to the program in Figure 9 (a). The computation of the attributes for each statement are given as follows:

1. First, we initialize $out(S_7) = \{\}$;

2. By (11): $ref(S_7)$={<B,(i=1, 100), i-1>, <C, (i=1, 100), i+1>, <D, (i=1, 100), i+1>};
   By (12): $def(S_7)$={<A, (i=1,100), i>};
   By (13): $in(S_7)$={<B, (i=1, 100), i-1>, <C, (i=1, 100), i+1>,<D, (i=1, 100), i+1>};

3. By (4), (8) and (9): $out(S_5) = out(S_6) = in(S_7)$;

Algorithm 1: Splitting an array reference.
Input: $u =< B, (i = l, h), i + c >$ and $v =< B', (i = l', h'), i + c' >$;
Output: A set of array references;

```
Split(u, v)
  If (B' ≠ B) Then return {u};
  Else
    case 1: l + c < l' + c' & h' + c' < h + c
      return {< B, (i = l, l' + c' − c − 1), i + c >,
                < B, (i = h' + c' − c + 1, h), i + c >};
    case 2: l' + c' ≤ l + c & h + c ≤ h' + c'
      return {};
    case 3: l + c < l' + c' < h + c < h' + c'
            (or l' + c' < l + c < h' + c' < h + c)
      return {< B, (i = l, l' + c' − c − 1), i + c >};
              (or {< B, (i = h' + c' − c + 1, h), i + c >})
    case 4: h + c < l' + c or h' + c < l + c
      return {u};
```

Figure 6: Operation *Split*.

Algorithm 2: Combining two array references.
Input: $u =< B, (i = l, h), i + c >$ and $v =< B, (i = l', h'), i + c >$
Output: A set of array references;

```
Widen(u, v)
  case 1: l < l' & h' < h
    return {< B, (i = l, h), i + c >};
  case 2: l' ≤ l & h ≤ h'
    return {< B, (i = l', h'), i + c >};
  case 3: l < l' < h < h' (or l' < l < h' < h)
      return {< B, (i = l, h'), i + c >};
              (or {< B, (i = l', h), i + c >})
  case 4: h < l' or h' < l
    return {u, v};
```

Figure 7: Operation *Widen*.

Algorithm 3: Intersecting two array references.

Input: $u =< B, (i = l, h), i + c >$, $v =< B, (i = l', h'), i + c >$
Output: A set of array references;

```
Narrow(u, v)
  case 1: l < l' & h' < h
    return {< B, (i = l', h'), i + c >};
  case 2: l' ≤ l & h ≤ h'
    return {< B, (i = l, h), i + c >};
  case 3: l < l' < h < h' (or l' < l < h' < h)
    return {< B, (i = l', h), i + c >};
              (or {< B, (i = l, h'), i + c >})
  case 4: h < l' or h' < l
    return {};
```

Figure 8: Operation *Narrow*.

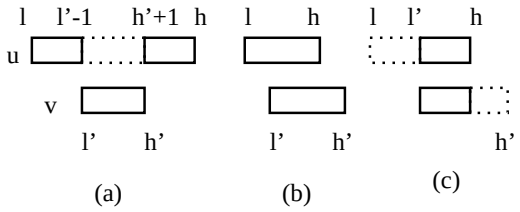

Figure 5: (a) Split operation; (b) Widen operation; (c) Narrow operation .

h'. Then we propagate only the common portion of the two references, $< B, (i = l', h, m), Source(i), Dest(i) >$, beyond the *If* structure. See Figure 5 (c).

In Section 3.3, we will study these cases in more details. In the rest of this section, we introduce the following definitions for the general form of array references:

Def. 1. Split(u,v): Given two array references u and v such that $Name_u = Name_v$, this function returns the portions of u that are not part of v.

Def. 2. Widen(u,v): Given two array references u and v such that $Name_u = Name_v$, $Source_u = Source_v$ and $Dest_u = Dest_v$, this function returns w with $D_w = D_u \cup D_v$.

Def. 3. Narrow(u,v): Given two array references u and v such that $Name_u = Name_v$, $Source_u = Source_v$ and $Dest_u = Dest_v$, this function returns w with $D_w = D_u \cap D_v$.

Def. 4. $U -_{<>} V$ is a set W such that
$W = \{u | u \in U \& (Name_v \neq Name_u \ \forall v \in V)\} \cup$
$\{Split(u,v) | u \in U \& v \in V \& Name_u = Name_v\}$

Def. 5. $U \cup_{<>} V$ is a set W such that
$W = \{u | u \in U \& (Name_v \neq Name_u \ \forall v \in V)\} \cup$
$\{v | v \in V \& (Name_u \neq Name_v \ \forall u \in U)\} \cup$
{Widen(u, v)— u$\in$ U& v$\in$ V& Name$_u$=Name$_v$&
$Source_u = Source_v \& Dest_u = Dest_v$}

Def. 6. $U \cap_{<>} V$ is a set W such that
$W = U \cap V \cup \{Narrow(u,v) | u \in U \&$
$v \in V \& Name_u = Name_v \&$
$Source_u = Source_v \& Dest_u = Dest_v\}$

In the next section, these operators are used to describe the data flow equations.

3.2 Data flow equations

The data flow analysis algorithm presented in this section is developed for the following language:

$$\begin{aligned} S &\longrightarrow S;\ S \mid If\ E\ then\ S\ else\ S \mid L \\ L &\longrightarrow Do\ i_1 = l_1, h_1, m_1; \ldots; i_n = l_n, h_n, m_n \\ &\quad ID_1[f(i_1, .., i_n)] = ..ID_2[g(i_1, .., i_n)].. \end{aligned}$$

This language is motivated by those data parallel programming languages in which array references are crucial for achieving parallelism [16] [8] [13]. For example, Fortran 90 array expression can be expressed using the above language. Therefore although the above language is simple, it expresses an important class of programs involving data level parallelism.

The data flow algorithm computes four synthesized attributes $ref(S)$, $def(S)$, $in(S)$, $out(S)$ for each construct S:

- $in(S) =\{ u | u$ is an array reference that is encountered before the array elements are redefined by a statement along each path starting at the point before $S\}$;
- $out(S) = \{u | u$ is an array reference that is encountered before the array elements are redefined by a statement along each path starting at the point after $S\}$;
- $ref(S) = \{u | u$ represents array elements that are used by $S\}$;
- $def(S) = \{u | u$ represents array elements that are defined by $S\}$; and

The data flow equations used to compute the above attributes are given as follows:

- $S \longrightarrow S_1;\ S_2$:

$$\begin{aligned} ref(S) &= ref(S_1) \cup_{<>} (ref(S_2) -_{<>} def(S_1)) && (1) \\ def(S) &= def(S_1) \cup_{<>} def(S_2) && (2) \\ in(S) &= in(S_1) && (3) \\ out(S_1) &= in(S_2) && (4) \\ out(S_2) &= out(S) && (5) \end{aligned}$$

Any array reference that is used by S_1 or used by S_2 but not defined by S_1 is considered as used by S. The array elements defined by S are defined by at least one of S_1 and S_2.

- $S \longrightarrow If\ E\ then\ S_1\ else\ S_2$:

$$\begin{aligned} ref(S) &= ref(S_1) \cap_{<>} ref(S_2) && (6) \\ def(S) &= def(S_1) \cup_{<>} def(S_2) && (7) \\ out(S_1) &= out(S) && (8) \\ out(S_2) &= out(S) && (9) \\ in(S) &= in(S_1) \cap_{<>} in(S_2) && (10) \end{aligned}$$

The set $ref(S)$ contains an array reference if it is used along both paths. An array reference in $def(S)$ is used along at least one of two paths.

- $S \longrightarrow Do\ i_1 = l_1, h_1, m_1; \ldots; i_n = l_n, h_n, m_n$
  $ID_1[f(i_1, .., i_n)] = \ldots ID_2[g(i_1, ..i_n)] \ldots$:

$$\begin{aligned} D &= \{ranges\ of\ i_j | (1 \leq j \leq n)\} \\ ref(S) &= \{< ID_2, D, g(i_1, \ldots, i_n), \\ &\quad f(i_1, \ldots, i_n) >\} && (11) \end{aligned}$$

```
Do i = 2, N                      Do i=2, N
   S(B[i-1], P_{i-1} → P_i)         S(B[i-1],C[i-1],P_{i-1} → P_i)
   R(B[i-1],P_i ← P_{i-1})          ...
   A[i] = B[i-1] + d             Do i=2, N
...                                 R(B[i-1],P_i ← P_{i-1})
Do i=2, N                           A[i]= B[i-1] + d
   S(C[i-1],P_{i-1} → P_i)       ...
   R(C[i-1],P_i ← P_{i-1})       Do i=2, N
   A[i] = A[i]+C[i-1]               R(C[i-1],P_i ← P_{i-1})
                                    A[i] = A[i]+C[i-1]
        (a)                              (b)
```

Figure 4: (a) A program with two communications; (b) After optimization.

P_{i-1} executes the receive instruction, it may be idle, waiting for $B[i]$ to arrive. In Figure 3 (b), the *receive* is separated from the *send* to increase the overlapping of communication and execution. Matching *send* and *receive* have to be separated in a manner which ensures that the execution of each *receive* matches the execution of a corresponding *send*.

(D) Combining Messages: In a distributed-memory system, there is a considerable setup overhead associated with the communication of each message. One way to reduce this overhead is to combine several small messages that need to be sent to the same destination into one large message so that the overhead of constructing and transmitting messages is amortized [15]. In Figure 4 (a), two values $B[i-1]$, and $C[i-1]$ are sent to P_i in the form of two messages, while in Figure 4 (b) they are combined into one message.

3 Data Flow Framework

In order to implement the communication optimizations, we propose a data flow analysis algorithm that collects the necessary information needed to perform the optimizations. For each array reference requiring communication, we determine all points in the program at which the communication can be performed. We compute this information by identifying the communication associated with each statement and propagating it in the backward direction along all execution paths in the program. An array reference nested within a loop may reference different array elements during each loop iteration. We represent these group of elements as a single entity. Groups of array elements that must be communicated are propagated. If definitions of array elements are encountered, the propagation of those elements is discontinued. We may encounter a definition which only defines some of the array elements. Thus a reference to a group of elements may have to be split. In addition, we may encounter repeated communication of the same array elements. In such a situation we must combine repeated references. In the subsequent sections we describe the representation of array references, and their propagation including *splitting* and *combining* operations.

3.1 Representation of array references

We need a concise notation to represent an array reference when performing data flow analysis for arrays. In [7], the authors compared several representations of array references by their precision and complexity. We use an extension of one of these notations and the extension allows us to keep track of the communication pattern, that is, which source processor should send what data to which consumer processor. We define an array reference as a tuple:

$$u =< Name, D, Source(i_1, \ldots, i_n), Dest(i_1, \ldots, i_n) >,$$

where $Name$ identifies the array that is referenced; D is the set of values that the index variables $i_1, \ldots, i_n$ take; $Source(i_1, \ldots, i_n)$ is a function indicating the array element that is used for the index point $(i_1, \ldots, i_n)$ in D and $Dest(i_1, \ldots, i_n)$ is a function representing the processor that uses the array element. For a m dimension array, $Source$ will produce a vector of m values. In general, both $Source$ and $Dest$ are functions of type: $I^n \rightarrow I^m$, where I is the set of integers. We call the *range* of the source function the *range* of the array reference. For an array reference u, $Name_u$, D_u, R_u, $Source_u$ and $Dest_u$ are the $Name$, the domain, the range, the source function and destination function of u, respectively.

Before we present the data flow equations, we define the operators *difference* ($-_{<>}$), *union* ($\cup_{<>}$) and *intersection* ($\cap_{<>}$) on sets of array references. These operators, which will be used in data flow equations, are different from simple set operators in that they require *splitting* and *combining* of array references. To explain the notation of *splitting* and *combining*, we consider a simple form of array reference: one dimensional array in a singly nested loop. In this case, the array reference can be represented as

$$u =< B, (i = l, h, m), Source(i), Dest(i) >$$

where l is the initial value of the loop variable i, h is its final value, and m is its increment. The algorithm will do backward data flow analysis, trying to locate the definition for B. During the processing, the following cases will arise:

1. Elements $B[Source(l')]$,.., $B[Source(h')]$ are defined and these definitions reach the reference u. If $l < l' < h' < h$, then we have to split u into two references with domain $(l, l'-1, m)$ and $(h'+1, h, m)$, respectively, and propagate them above the definitions. See Figure 5 (a).

2. Another reference $v =< B, (i = l', h', m), Source(i), Dest(i) >$, is encountered. If $l < l' < h < h'$, then we can combine u and v into one array reference $< B, (i = l, h', m), Source(i), Dest(i) >$. See Figure 5 (b).

3. The reference u is in one branch of an *If* structure, $v =< B, (i = l', h', m >, Source(i), Dest(i) >$ is in the other branch and $l < l' < h <$

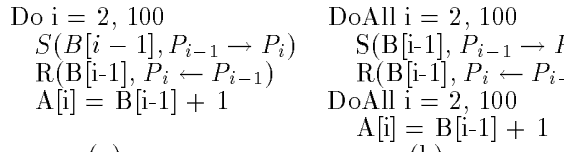

Figure 1: (a) A forced sequential loop; (b) After optimization.

The remainder of the paper is organized as follows. In Section 2, we identify the communication optimizations. In Section 3, we present the data flow framework for collecting the information necessary to perform the optimizations. In Section 4, we provide an integrated communication optimization algorithm. In Section 5, we point out some extensions of the basic data flow analysis algorithm. We discuss related work in Section 6 and give a summary in Section 7.

2 Communication Optimizations

We assume that a serial program is translated for SPMD execution, using the owner computes rule. In this approach each processor examines the statements in the program sequentially and for each statement it takes one of the following actions: (a) performs the operation indicated by the statement if it owns the data element whose value is being computed. Non-local operands are received from other processors; (b) sends a local data to the processor that needs the data to execute the statement; or otherwise (c) skips the statement. In this approach communication instructions are introduced by the compiler. The communication primitives are non-blocking *send* and blocking *receive*. It is assumed that the instruction *send* transmits both the name and the value of the data to the destination. We will use the following notations for *send* and *receive* instructions:

S(name, $P_s \rightarrow P_d$)≡ P_s Sends {name=val(name)} to P_d,
R(name, $P_d \leftarrow P_s$)≡ P_d Receives {name=val(name)} from P_s.

If the same variable is sent twice with different values, they are delivered to the destination in the same order as they are sent. We consider the following optimization problems:

(A) Avoiding Sequentialization Caused by Communication: A key problem with the basic translation techniques for SPMD execution is that the data-owner processor sends the data to the data-consumer processor at the same point in the program at which the data is used [1]. As illustrated in [1] [6], this approach can sequentialize the execution of otherwise parallel code. For example, consider the loop in Figure 1 (a). In order to simplify the presentation, we assume that data array be distributed such that $A[i]$ and $B[i]$ are stored in the local memory of P_i. The loop in Figure 1 (a) will be executed sequentially in SPMD mode. Processor P_{i-1} must finish executing the $(i-1)$th iteration before sending the value of $B[i-1]$ to P_i. Thus processor P_i, which is to execute the ith iteration, is kept waiting. If $B[i-1]$ is sent to P_i before the loop starts execution, as shown in Figure 1 (b), then all iterations can be executed in parallel. In general, we would like to send data as early as possible, since the earlier the data is sent the less the chance that the consumer processor will be delayed and thus the better the chance that all processors can execute the program in parallel.

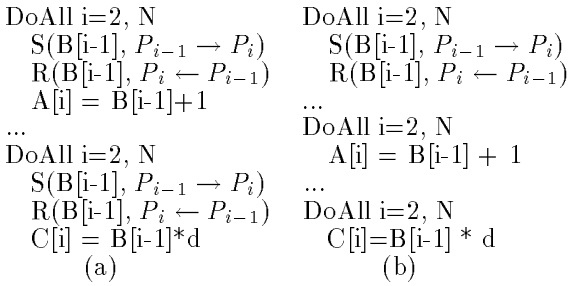

Figure 2: (a) The program with redundant communication; (b) After optimization.

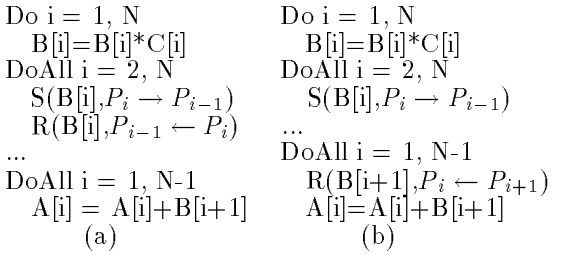

Figure 3: (a) Avoiding sequentialization; (b) Separating send and receive.

(B) Avoiding Redundant Communication:[6] Sometimes a value is used more than once by a processor. For example, in Figure 2 (a) $B[i-1]$ is used twice by processor P_i. Assuming that there is no definition for $B[i-1]$ between the two references, the value $B[i-1]$ may only be sent once. We want to identify this kind of redundant communications and eliminate all but the first communication, as shown Figure 2 (b).

(C) Overlapping Communication with Computation: In SPMD mode, *send* and *receive* are inserted at the same point in the program [1]. Since the *send* is non-blocking and the *receive* is blocking, the processor that executes a *receive* instruction will be blocked until the data arrives. Thus, inserting *send* and *receive* at the same point is likely to cause delays during execution since the latency of data communication in a distributed-memory system is large. This delay can be reduced by maximizing the overlapping of communication and computation. Such overlap may be achieved by inserting the *send* at an earlier point than the *receive*. Consider the example of Figure 3 (a), in which we insert both *send* and *receive* instruction at the same point (after $B[i]$ is defined). When processor

Compilation Techniques for Optimizing Communication on Distributed-Memory Systems

Chun Gong
gong@cs.pitt.edu

Rajiv Gupta
gupta@cs.pitt.edu

Rami Melhem
melhem@cs.pitt.edu

Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

Communication overhead can significantly impact the performance during parallel execution of programs on distributed-memory systems. In this paper we describe optimizations that reduce communication overhead and avoid execution time delays caused by interprocessor communication. The optimizations include: (a) avoiding sequentialization caused by communication; (b) avoiding redundant communication; (c) overlapping the communication and the computation; and (d) combining several small messages sent to the same destination into one large message. We develop a data flow framework for collecting information necessary to apply the above optimizations to programs being compiled for single-program, multiple-data *(SPMD) execution on a distributed-memory system. We are able to deal with the tradeoff between conflicting optimizations through this unifying data flow framework.*

1 Introduction

While a distributed-memory system can exploit massive parallelism due to its scalability, it is difficult to program such an architecture. The programmer must explicitly distribute data and must program the transfer of data among the processors. One approach considered by researchers provides a combination of language extensions and compilation techniques for programming such systems [1] [9] [10] [15]. In this approach, the user specifies the distribution of data by using language extensions and the compiler translates the program for *single-program multiple-data* (SPMD) execution. One approach to translation yields a program in which a processor executes a statement only if a data value computed by the statement resides at the processor. If a processor needs a data from a nonlocal memory, a *receive* instruction must be executed by the processor and a *send* instruction must be executed by the processor that owns the data. These *receive* and *send* instructions are automatically inserted by the compiler.

The communication overhead latency is of primary concern during the SPMD execution of programs. First, the long communication latency may cause a processor to idle while waiting for the data it needs. Second, some communication schemes may sequentialize the execution of an otherwise parallel computation. Therefore for the SPMD approach to work effectively, optimizations must be performed to reduce the communication overhead. In this paper we develop compile-time techniques for optimizing this overhead. The goals of the optimizations are:

1. Avoiding sequentialization caused by communication;
2. Avoiding redundant communication;
3. Overlapping of communication and computation; and
4. Combining small messages sent to the same destination into larger messages.

Significant work has been done in optimizing the communication [1] [6] [4] [11]. In this paper, we propose to perform all of the above optimizations in a unifying framework. This framework allows us to deal with the tradeoff between conflicting optimizations. We develop a data flow framework for collecting the information needed to perform the above optimizations. We focus on optimizing the communication resulting from references to distributed arrays. More specifically, a data flow analysis algorithm, which propagates portions of arrays, is used to determine points in the program at which communication can be performed. Using this information, optimization algorithms select appropriate communication points to achieve the above optimizations. Since arrays are used very often in scientific applications, significant performance improvement can be expected by optimizing array references. In order to develop efficient data flow analysis algorithms we treat an array reference as an integrated unit. All array elements corresponding to a reference are represented by a single pattern rather than being represented individually.