Combining Dependence and Data-Flow Analyses to Optimize Communication

Ken Kennedy

Nenad Nedeljković

CRPC-TR94484-S

September, 1994

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

A modified version of this paper will appear in the Proceedings of the 9th International Parallel Processing Symposium, IPPS '95.

Combining Dependence and Data-Flow Analyses to Optimize Communication *

Ken Kennedy
ken@rice.edu

Nenad Nedeljković
nenad@rice.edu

Center for Research on Parallel Computation
Department of Computer Science, Rice University

Abstract

Reducing communication overhead is crucial for improving the performance of programs on distributed-memory machines. Compilers for data-parallel languages must perform communication optimizations in order to minimize this overhead. In this paper, we show how to combine dependence analysis, traditionally used to optimize regular communication, and a data-flow analysis method originally developed to improve placement of irregular communication. Our approach allows us to perform more extensive optimizations — message vectorization, elimination of redundant messages, and overlapping communication with computation. We also present preliminary experimental results that demonstrate the benefits of the proposed method.

1 Introduction

Distributed-memory machines are becoming widely accepted as the most promising way to achieve high performance computing. They scale to hundreds, and even thousands of processors, thus providing unmatched computing power. However, harnessing this power is an extremely difficult task. Since these computers lack global address space, communication has to be inserted whenever a processor needs to access non-local data. Managing this communication explicitly is a very tedious and error-prone process.

In recent years significant efforts have been made to develop programming languages that provide shared address space in order to free the programmer from the burden of manipulating messages directly. Data-parallel languages, such as Fortran D [17], Vienna Fortran [5], and High Performance Fortran (HPF) [19], provide directives for the programmer to specify how data should be distributed among processors. Using the distribution directives, compilers for these languages (which are often source-to-source translators) generate SPMD-style

*This work was supported in part by ARPA contract DABT63-92-C-0038. and NSF Cooperative Agreement Number CCR-9120008. The content of this paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

programs in which the computation is partitioned across processors and the communication inserted wherever necessary.

The main obstacle in achieving high performance when running these programs on distributed architectures is the fact that the cost of interprocessor communication in modern multicomputers is significantly higher than the cost of accessing local data. Therefore, it is extremely important to reduce the amount of communication inserted by the compiler. In the Fortran D compiler prototype developed at Rice University, communication resulting from regular array references (those references where subscripts are linear combinations of loop induction variables) is optimized based primarily on dependence analysis [16]. The most important optimizations performed are *message vectorization* (which hoists communication out of loops thus combining multiple single-element messages into a single vectorized message) and *message coalescing* (which combines messages resulting from different references to the same array) [18]. However, the effectiveness of optimizations is limited by the fact that most of the analysis is performed for a single loop nest at a time, and very little is done to optimize communication across arbitrary control flow.

On the other hand, communication caused by irregular array references (those where subscripts are not linear combinations of loop induction variables, but for example array lookups) is optimized via the powerful GIVE-N-TAKE code placement framework [14]. Although this framework provides global analysis on the control flow graph, its current limitation is the treatment of arrays as indivisible units. Because of this, it does not take advantage of certain optimization opportunities that come from the compile-time knowledge about array references.

In this paper we show how to combine dependence analysis, which provides the information about array elements and sections accessed, and data-flow analysis based on the GIVE-N-TAKE framework, to propagate this information across arbitrary control flow. The combined approach allows us to perform more extensive optimizations than either of the two components would do on its own.

There have been several attempts to use data-flow analysis in order to optimize communication [8, 7, 11]. Most of these efforts have focused on extending the existing data-flow analysis methods to work with some form of array section descriptors. In contrast, our data-flow analysis uses bit vectors (with each bit representing an array portion) and is thus likely to be more efficient. While this approach could result in a less precise analysis, we maintain high level of precision by examining the relationships among array portions when initializing the data-flow framework. Our preliminary experiments indicate that this precision is satisfactory for data-parallel scientific kernels, and that the proposed method is useful for reducing communication cost.

The remainder of this paper is organized as follows: In Section 2 we provide further motivation for our work by discussing the benefits and drawbacks of the two current approaches and outlining possible improvements. Our method for combining dependence and data-flow analyses is described in detail in Section 3, and preliminary experimental results are presented in Section 4. We discuss related work in Section 5 and conclude in Section 6 by summarizing our contributions and indicating directions for future work.

2 Motivation

In this section we describe the analyses and optimizations performed in the current Fortran D compiler prototype. We split our discussion in two parts. First, we briefly describe the pros and cons of the dependence-based approach, which is used to optimize communication caused by regular array references. Second, we give an outline of the data-flow framework used to analyze irregular array accesses and present some problems that arise when this analysis is applied to regular references.

2.1 Optimizing Regular Communication

We call an array reference regular if its subscripts are linear combinations of loop induction variables of the loops that enclose the reference. The most important optimization for these references is *message vectorization*. It uses the level of loop-carried true dependences to decide if the communication can be hoisted out of the loop, in which case many small messages are replaced with one large message. This algorithm was first described in [3] and [6], and its implementation in the Fortran D compiler is detailed in [20].

In order to avoid communicating redundant data, the compiler applies *message coalescing*, which combines messages for different references to the same array. In the absence of data-flow analysis, this optimization is performed only within a single loop nest, and thus many opportunities for eliminating redundant messages may be missed.

Overlapping communication and computation is an important technique used to improve the performance of programs on distributed-memory machines. The Fortran D compiler tries to achieve this overlap through *vector message pipelining*, an optimization that moves SEND and RECV statements towards their definitions and uses respectively. However, the support for this optimization in the current compiler prototype is very limited.

The program in Figure 1 illustrates the analysis and optimization performed for regular references. The original Fortran D program with data decomposition and distribution directives is shown on the left. Since the number of processors is 5, each processor will get 10 elements of each of the arrays `a`, `b`, and `c`. Based on this distribution, the compiler partitions the computation by applying the *owner computes* rule and inserts necessary communication, as shown on the right side. For example, statement COMM $1 \xrightarrow{b(11)} 0$ means that the array element `b(11)` is sent from processor `1` to processor `0`. Although the compiler performs loop bound reduction when translating the Fortran D program into SPMD node program, in the interest of readability we show all the loops and communication in terms of global indices.

In our example, communication ③ of the elements of array `b` is clearly redundant, since these same elements have already been communicated in ①, in order to satisfy the non-local references to `b` in the first `DO` loop. Because the compiler performs message coalescing only within a single loop nest, it is not able to remove this redundant communication.

Somewhat more complicated is the case of communication ④. This communication is partially redundant, because it is only redundant if the `THEN` branch of the `IF` statement is taken, in which case the same elements of array `a` will already have been communicated in ②. Because of this partial redundancy, it is desirable to move communication ④ inside the `ELSE` branch, just after the `DO` loop that defines elements of `a`, but the compiler does not perform the analysis necessary for this kind of optimization.

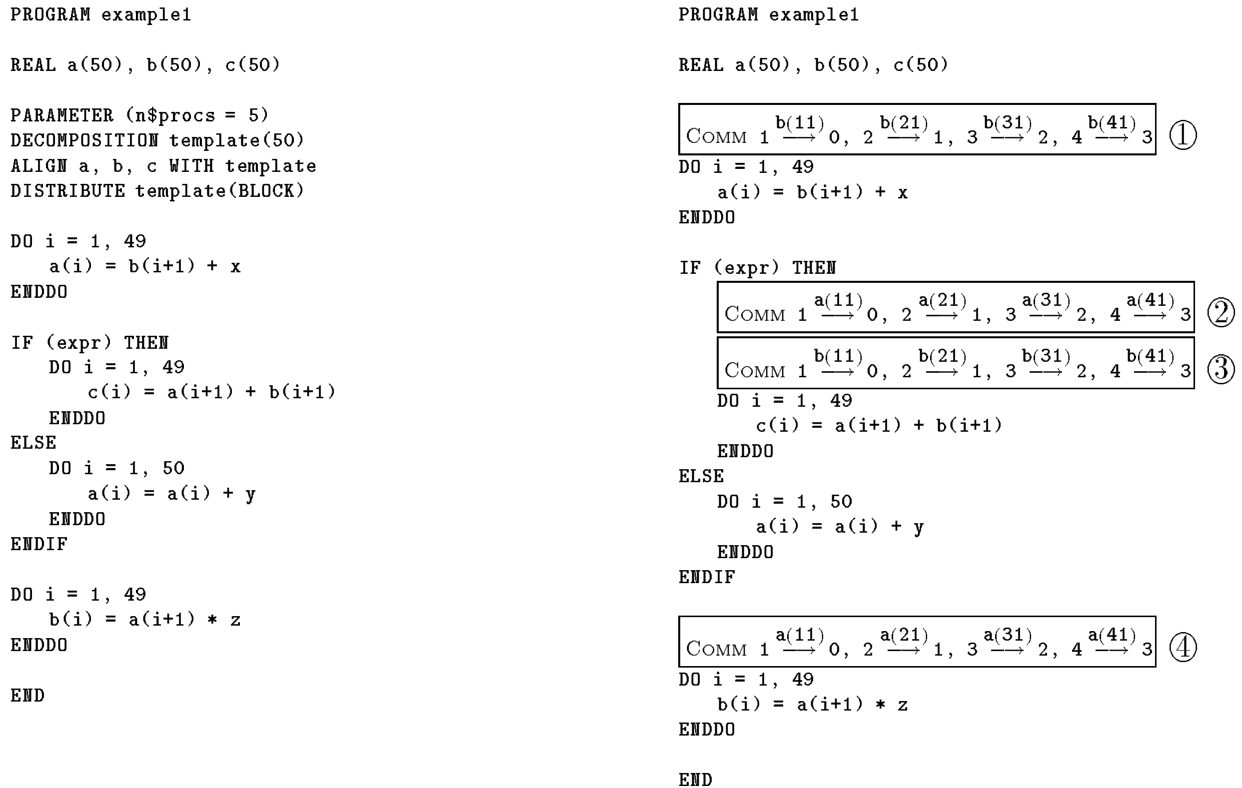

Figure 1 Communication placement based on dependence analysis.

2.2 Give-N-Take Code Placement Framework

Most common examples of irregular array references are those in which an array subscript itself is a reference to an indirection array. In these cases it is hardly possible to extract any significant compile-time knowledge about array sections accessed, and therefore dependence analysis is of little use. Instead, the Fortran D compiler prototype uses the analysis provided by the GIVE-N-TAKE code placement framework [14]. This very general framework uses a producer-consumer concept where references to potentially non-local data are viewed as consumption, and the communication that fetches the data represents the production whose placement needs to be determined. Additionally, the framework also takes into account that data may also be destroyed, i.e., redefined, or given for free through side effects.

For each node in the control-flow graph, initial variables of the framework describe consumption (TAKE_{init}), destruction (STEAL_{init}), and side effects (GIVE_{init}) at the corresponding location in the program. The algorithm for data-flow analysis then propagates this information globally, by evaluating the complex equations described in [14]. The result of the analysis is given by output variables that indicate where the production of data should be placed. If we are only trying to satisfy references to non-local data, the production will be in the form of global READ (GATHER) statements. If we relax the owner computes rule to allow definitions of non-owned data, than we also need to find placement for global WRITE

(SCATTER) statements, which will indicate when these data should be sent back to their owners.

Optimizations performed by the framework include global elimination of redundant production (communication) and latency hiding achieved by splitting each production (READ) in such a way that its start (READ_{Send}) is placed as early as possible and its end (READ_{Recv}) as late as possible.

Although the GIVE-N-TAKE framework is in principle applicable to analysis of regular references, its current implementation has several limitations, most of which come from the fact that the initial focus was on the analysis of array accesses with irregular subscripts. For example, in the READ problem TAKE_{init} set for each node includes all the array portions referenced at that point in the program [12]. For irregular problems, determining which of these references require communication is done at run time through calls to PARTI/CHAOS library routines (GATHER) [4]. However, in regular codes, it is often possible for the compiler to extract much more static information and determine at compile time which references are non-local.

When comparing two portions of the same array, the compiler must assume the most conservative facts if these portions are accessed via irregular subscripts. For example, when deciding whether communication of an array portion can be hoisted across a definition of the other portion, we need to find out whether these two portions have non-empty intersection. If these are irregular portions of the same array, at compile time we must assume that they might interfere with each other, since otherwise we might place communication in such a way that the data are sent before they are defined. Similarly, for two irregular array portions representing data to be communicated, it is not safe to assume that either one can be subsumed by the other, unless the portions are identical. In contrast, regular array portions often provide enough information for the compiler to answer some of these questions more precisely.

The example in Figure 2 shows the results of applying the analysis developed for irregular problems to the code in which array references are regular. Communication ①, which performs a global READ for `a(2:50)`, is delayed until after the execution of the first `DO` loop. The reason for this is that the GIVE-N-TAKE framework conservatively assumes that the definition of array elements in the first `DO` loop interferes with the data that is communicated in ①, just because they both use the same array `a`. Furthermore, even if the compiler compared regular array sections `a(2:50:2)` (defined in the first loop) and `a(2:50)` (referenced in the second loop), it would find out that these two have non-empty intersection. The communication would therefore stay at the same place, since the compiler passes the whole array portion referenced to the run-time library (the READ statement would actually be converted into a call to the GATHER routine) and lets the run-time system decide which accesses really require communication. However, if we look more closely at this communication, we can see that array elements `a(11, 21, 31, 41)`, which participate in it, are actually not defined in the first loop. This means that the SEND part of communication ① could be moved before the first `DO` loop, and thus the message transfer time could be overlapped with the execution of that loop.

In the last loop nest communication ② is correctly hoisted out of both loops, but the communication ③ is not. The reason for this is that the framework does not analyze array references on a per element basis, but instead assumes that the same portion of array `c` is

```
PROGRAM example2

REAL a(50), b(50), c(50)

PARAMETER (n$procs = 5)
DECOMPOSITION template(50)
ALIGN a, b, c WITH template
DISTRIBUTE template(BLOCK)

DO i = 2, 50, 2
   a(i) = x
ENDDO

DO i = 1, 49
   b(i) = a(i+1) * y
ENDDO

DO i = 1, 49
   DO j = 1, 10
      c(i) = c(i) + c(i+1) * b(j)
   ENDDO
ENDDO

END
```

```
PROGRAM example2

REAL a(50), b(50), c(50)



DO i = 2, 50, 2
   a(i) = x
ENDDO

[READ SEND/RECV a(2:50)] ①
DO i = 1, 49
   b(i) = a(i+1) * y
ENDDO

[READ SEND/RECV b(1:10)] ②
DO i = 1, 49
   DO j = 1, 10
      [READ SEND/RECV c(i+1)] ③
      c(i) = c(i) + c(i+1) * b(j)
   ENDDO
ENDDO

END
```

Figure 2 Communication placement based on GIVE-N-TAKE analysis.

both defined and used within the loop nest. However, the method based on the level of loop-carried true dependences (described in the previous section) would place this communication at the same place as ②.

3 Combined Approach

In the previous section we have exposed the shortcomings of each of the two separate analyses performed by the Fortran D compiler. We now show how to combine these two analyses in an effort to optimize communication placement for regular computations.

3.1 Initial Communication Annotations

As a first step we perform dependence analysis for all regular references. Results of this analysis are then used for communication vectorization and coalescing within each loop nest. By doing this, we are able to avoid the problem that the GIVE-N-TAKE framework has with hoisting communication out of the loop in the presence of anti dependences.

In contrast to the existing Fortran D compiler, we do not yet generate actual messages, but instead each node in the control-flow graph is annotated with sets of array elements that need to be communicated (COMM) at that point in the program. Although messages that will eventually be generated from these annotations are created using indices local to each processor, at this point we still represent communication sets in terms of global indices from the original Fortran D program.

The current Fortran D compiler prototype uses *regular section descriptors* (RSDs) to represent the sets of array elements that need to be communicated [15]. These RSDs are augmented to handle simple forms of boundary conditions [20]. While this representation is sufficient for one-dimensional `BLOCK` and `CYCLIC` distributions currently supported by the compiler, a more general representation is necessary for communication sets that arise with `BLOCK_CYCLIC` and multi-dimensional distributions. However, our approach is independent of the representation used for communication sets, as long as this representation supports some basic operations; we should be able to determine if two given sets intersect, and if one is a subset of the other. Naturally, the increased precision of these operations is likely to also increase the precision of the overall analysis.

It is important to note that for each communication annotation we need not only record the set of array elements that should be communicated, but also keep track of the processors participating in the communication. This information is also produced by the current compiler, which classifies communication according to the patterns of references that cause it.

3.2 Computing Data-Flow Variables

After the communication sets are computed based on dependence analysis, we use these sets to define the input variables for the GIVE-N-TAKE framework. Because we assume that the owner computes rule is used to partition the computation, we only need to solve global READ problem. Since the owner computes rule does not allow definitions of non-owned data, we only need to determine the placement of communication necessary to satisfy non-local references. In order to do this we initialize the input variables for each node as follows:

$$\begin{aligned} \text{READ.TAKE}_{init} &:= \text{COMM}, \\ \text{READ.STEAL}_{init} &:= \mathit{affects}(\text{DEF}), \\ \text{READ.GIVE}_{init} &:= \mathit{contains}(\text{COMM}). \end{aligned}$$

Each variable represents a bit vector in which each bit position corresponds to an array portion. Consumption that needs to be satisfied at a certain node in the control-flow graph (TAKE_{init}) is given by the COMM set for that node — it includes all array portions that should be communicated at that point, as determined by dependence analysis. For example, all COMM sets shown in Figure 1, would go into TAKE_{init} sets for their corresponding locations in the program. In contrast to the original analysis of irregular problems [12], where TAKE_{init} sets contain all referenced portions of distributed arrays, our consumption sets will be smaller whenever the elements to be communicated can be determined at compile time. More precise consumption sets will often open ground for more optimization opportunities, as shown by the example in Figure 2. Furthermore, since our data-flow variables include only those bits that represent array portions for which communication is required (instead of all array portions referenced in the program), this will make bit vectors shorter, and therefore the analysis will be faster.

STEAL_{init} set for each node includes all COMM sets in the program that could be *affected* by definitions at that point in the control-flow graph (DEF). In other words, if an array portion that needs to be communicated somewhere in the program has non-empty

intersection with the array portion defined at the current location, then the former belongs to the STEAL_{init} set of the current control-flow node. These sets are used to prevent the moving of communication statements across definitions of data that is to be communicated. For example, in Figure 1 COMM set ④ would be *stolen* by the definition of array portion `a(1:50)` in the `ELSE` branch of the `IF` statement, which prevents this communication to be hoisted before the whole `IF` statement.

GIVE_{init} sets are used to eliminate redundant and partially redundant communication. If an array portion belongs to a COMM set for a control-flow node, then all array portions that should be communicated elsewhere in the program, and that are fully *contained* in the given portion, belong to the GIVE_{init} set for that node. For example, communication ③ in Figure 1 would be *given* by the communication ① of the same array. By initializing GIVE_{init} sets this way, we can eliminate messages not only when array portions are identical to those already communicated (as is the case in Figure 1), but also when they are subsets of previously communicated data.

As mentioned in Section 3.1, when checking if one communication subsumes another, it is not enough to look only at the array elements communicated, but we also need to take into account processors participating in the communication. For example, given a shift communication

$$\text{COMM}\ \ 1 \xrightarrow{\texttt{a(11)}} 0,\ \ 2 \xrightarrow{\texttt{a(21)}} 1,\ \ 3 \xrightarrow{\texttt{a(31)}} 2,\ \ 4 \xrightarrow{\texttt{a(41)}} 3$$

and a broadcast communication

$$\text{COMM}\ \ 1 \xrightarrow{\texttt{a(11)}} 0,\ 2,\ 3,\ 4$$

the former does not subsume the latter although the set of elements communicated in the broadcast (`a(11)`) is fully contained within the set of elements communicated in the shift (`a(11, 21, 32, 41)`). To see this, it is enough to note that the shift communication makes element `a(11)` available only at processor `0` (and owning processor `1`), while the broadcast makes this element available at all 5 processors.

Once the input variables are initialized, we proceed with the GIVE-N-TAKE analysis as described in [14]. All lattice operations needed to evaluate data-flow equations are performed on bit vectors. It would be possible to modify the framework so that it performs lattice operations on array portions themselves, instead of using just bits that represent those array portions. Although this approach could in some cases be more precise, we chose not to do so for two reasons. First, our method is more efficient because we only look at the relationships among array sections when initializing the framework, and during the data-flow analysis we perform simple logical operations on bit vectors. On the other hand, evaluating intersection, union and difference of array portions can be quite non-trivial and possibly time consuming (depending on the representation used and the precision desired). The second reason for our approach is that it allows easy integration with already existing analysis for irregular problems, where compile-time knowledge about array portions is insignificant and an effort to perform lattice operations on these portions would prove fruitless.

```
PROGRAM example3
REAL a(50), b(50), c(50), d(50)

[COMM 1 --b(11)--> 0, 2 --b(21)--> 1, 3 --b(31)--> 2]  ①
DO i = 1, 30
   a(i) = b(i+1) + x
ENDDO

a(31) = b(31)

IF (expr) THEN
   [COMM 1 --a(11)--> 0, 2 --a(21)--> 1]  ②
   DO i = 1, 24
      c(i) = c(i) + a(i+1)
   ENDDO
ELSE
   [COMM 1 --b(11)--> 0, 2 --b(21)--> 1]  ③
   DO i = 1, 24
      c(i) = c(i) - b(i+1)
   ENDDO
ENDIF

[COMM 1 --a(11)--> 0, 2 --a(21)--> 1]  ④
DO i = 1, 24
   b(i) = a(i+1) * y
ENDDO

[COMM 1 --a(11)--> 0,2 --a(21)--> 1,3 --a(31)--> 2,4 --a(41)--> 3]  ⑤
[COMM 1 --c(11)--> 0,2 --c(21)--> 1,3 --c(31)--> 2,4 --c(41)--> 3]  ⑥
[COMM 0 --d(1:10)--> 1,2,3,4]  ⑦
DO i = 1, 49
   DO j = 1, 10
      c(i) = c(i) + c(i+1) * d(j) + a(i+1)
   ENDDO
ENDDO
END
```

```
PROGRAM example3
REAL a(50), b(50), c(50), d(50)

[SEND 0 --d(1:10)--> 1,2,3,4]  ①
[SEND/RECV 1 --b(11)--> 0, 2 --b(21)--> 1, 3 --b(31)--> 2]  ②
DO i = 1, 30
   a(i) = b(i+1) + x
ENDDO

[SEND 1 --a(11)--> 0, 2 --a(21)--> 1]  ③

a(31) = b(31)

IF (expr) THEN
   [RECV 1 --a(11)--> 0, 2 --a(21)--> 1]  ④
   DO i = 1, 24
      c(i) = c(i) + a(i+1)
   ENDDO
ELSE
   DO i = 1, 24
      c(i) = c(i) - b(i+1)
   ENDDO
   [RECV 1 --a(11)--> 0, 2 --a(21)--> 1]  ⑤
ENDIF

[SEND 1 --c(11)--> 0,2 --c(21)--> 1,3 --c(31)--> 2,4 --c(41)--> 3]  ⑥
DO i = 1, 24
   b(i) = a(i+1) * y
ENDDO

[SEND/RECV 1 --a(11)--> 0,2 --a(21)--> 1,3 --a(31)--> 2,4 --a(41)--> 3]  ⑦
[RECV 1 --c(11)--> 0,2 --c(21)--> 1,3 --c(31)--> 2,4 --c(41)--> 3]  ⑧
[RECV 0 --d(1:10)--> 1,2,3,4]  ⑨
DO i = 1, 49
   DO j = 1, 10
      c(i) = c(i) + c(i+1) * d(j) + a(i+1)
   ENDDO
ENDDO
END
```

Figure 3 Left: Initial communication annotations based on dependence analysis. Right: Optimized placement of SENDs and RECVs based on data-flow analysis.

3.3 Optimized Communication Placement

The example shown in Figure 3 illustrates the results of our combined analysis. Redundant and partially redundant communication is eliminated, and SENDs are separated from their corresponding RECVs by being moved as far up in the control-flow graph as allowable (up to the precision of array portion analysis).

Alignment and distribution directives from the original Fortran D program are not shown

in Figure 3, but instead it is assumed that all arrays are perfectly aligned and `BLOCK`-distributed over 5 processors. As in our previous examples, loop bounds are shown in their original form, even though they will be reduced when the compiler performs computation partitioning. On the left side we show the program with communication sets inserted based on dependence analysis.

In contrast to Figure 2, where the GIVE-N-TAKE framework did not allow hoisting of communication ③ out of the loop, dependence analysis is sufficient to discover that similar communication ⑥ on the left side of Figure 3 can be performed before the last loop nest. After this problem is solved, and TAKE sets are initialized to contain only array portions that might require communication, annotations to the control-flow graph are propagated to achieve the placement of SENDs and RECVs, as shown on the right side.

Statement ③ on the left has been eliminated, since the data that it would communicate are the subset of the data that had already been communicated before the first `DO` loop (statement ① on the left, corresponding to ② on the right).

Partial redundancy of communication ④ on the left has been removed by moving this communication into the `ELSE` branch of the `IF` statement. Furthermore, the SEND statement for this communication has been combined with the SEND corresponding to communication ② on the left. This SEND statement (③ on the right) has then been moved as far as possible from its matching RECVs (④ and ⑤ on the right) in order to overlap communication with computation.

Similarly, separation of SENDs and RECVs has also been done for communication ⑥ (SEND – ⑥, RECV – ⑧), and communication ⑦ (SEND – ①, RECV – ⑨). It should be noted that SEND statement ③ that initiates the communication of elements of array `a` has been moved across the definition of an element of the same array, since the array section defined (`a(31)`) does not intersect the array section communicated (`a(11, 21)`). However, statement ③ could not be moved any further up, since the first `DO` loop defines the array elements that are communicated.

3.4 Local Scheduling

A seemingly trivial, but in practice important implementation detail is the way that messages are scheduled at each node in the control-flow graph. Although the resulting annotations are grouped based on whether the required communication statement is SEND, RECV, or SEND/RECV, the following simple rule should be used when actually generating code from these annotations. At each point in the program,

1. Send data that require SEND/RECV at the same control-flow node,
2. Send data that require SEND at this node, but are received elsewhere,
3. Receive data that only have RECV at this point, but are sent at some other node, and
4. Receive data sent in step 1 in the same order that they were sent, in an attempt to achieve at least some communication/computation overlap even at the local level.

3.5 Discussion

While the optimizations described in Section 3.3 represent an improvement over what the current Fortran D compiler would do, there is still some communication redundancy left. A part of communication ⑦ on the right side of Figure 3 is redundant, because the communication $\{1 \xrightarrow{\texttt{a(11)}} 0,\ 2 \xrightarrow{\texttt{a(21)}} 1\}$ had already taken place (SEND – ③, RECV – ④, ⑤), and array elements `a(11)` and `a(21)` were not redefined. Therefore, only the communication $\{3 \xrightarrow{\texttt{a(31)}} 2,\ 4 \xrightarrow{\texttt{a(41)}} 3\}$ is necessary. Our framework as presented fails to find this redundancy. The reason for this is our treatment of array portions through their representing bits in bit-vector based flow analysis. We do not analyze relationships among array portions beyond the initial determination of whether a definition of one interferes with a communication of the other (STEAL_{init}) and whether a communication is fully contained in the other (GIVE_{init}).

It is not clear whether partly redundant[1] communication is a commonly occurring case in real programs. Furthermore, elimination of these redundancies does not necessarily reduce the total execution time. In our example, the cost of communicating $\{1 \xrightarrow{\texttt{a(11)}} 0,\ 2 \xrightarrow{\texttt{a(21)}} 1,\ 3 \xrightarrow{\texttt{a(31)}} 2,\ 4 \xrightarrow{\texttt{a(41)}} 3\}$ should be practically the same as the cost of $\{3 \xrightarrow{\texttt{a(31)}} 2,\ 4 \xrightarrow{\texttt{a(41)}} 3\}$, since all of the individual messages can be transferred in parallel. If, however, we wanted to remove even partly redundant communication, several approaches could be used to address this issue.

A simple way is to find, for each communication statement, all other places in the program where portions of the same array are communicated and from which there is a control-flow path to the statement under consideration. Using this information we could split the array portion communicated into parts corresponding to array portions in reaching communications. (This is similar to splitting in [7], but we only perform splitting when initializing the data-flow framework.) Since we are only interested in control-flow reachability without taking array kill information into account, this approach does not require array data-flow analysis. In our example on the left side of Figure 3, communication ④ reaches communication ⑤, which would cause the latter to be split, leading to the elimination of partly redundant communication described above.

If we are willing to pay the full price of array data-flow analysis, such as that described in [9], better precision could be achieved by using reaching array section definitions as the basis for splitting. In that case, communication ⑥ on the left side of Figure 3 would also be split (into $\{1 \xrightarrow{\texttt{c(11)}} 0,\ 2 \xrightarrow{\texttt{c(21)}} 1\}$ and $\{3 \xrightarrow{\texttt{c(31)}} 2,\ 4 \xrightarrow{\texttt{c(41)}} 3\}$), and the part SEND $\{3 \xrightarrow{\texttt{c(31)}} 2,\ 4 \xrightarrow{\texttt{c(41)}} 3\}$ would be moved all the way to the beginning of the program.

We intend to investigate how frequently partly redundant communication occurs in scientific codes, and what the benefits of eliminating it are. Although our current focus is on the efficient analysis with possible sacrifice of precision, it is possible that our data-flow analysis will be extended to work with array portions directly. In this light we are also investigating new array section representation that would be general enough to handle communication sets that arise from `BLOCK_CYCLIC` and multidimensional distributions, and yet allow efficient lattice operations needed for data-flow analysis.

[1] Note the difference between partially redundant communication, which is redundant on some control-flow paths, and partly redundant communication, for which only some of the participating messages are redundant.

4 Preliminary Experience

The implementation of the GIVE-N-TAKE framework that provides bit-vector based analysis has been described in [12]. We have modified the existing framework to include the support for handling array portions that can be represented with RSDs. Although the framework can now take some advantage of compile-time knowledge about array elements accessed, full propagation of dependence-based communication analysis has not yet been implemented. However, we were able to run an experiment to measure the potential benefits of our approach.

In our experiment we used LIVERMORE 18 explicit hydrodynamics kernel and SHALLOW weather prediction program written by Paul Swartztrauber, National Center for Atmospheric Research (NCAR). Both benchmarks are highly data-parallel and significant speedups were reported when they were compiled with the existing Fortran D compiler [20]. However, hand-coded versions were still up to 25% faster, with most of the difference coming from eliminating redundant messages and increasing the overlap of communication with computation — exactly the optimizations that we propose to automate. (Although hand-coded versions of some benchmarks described in [20] were up to 50% faster (DGEFA and ERLEBACHER), these improvements were due to aggressive program transformations [1], which are beyond the scope of this paper.)

After translating original Fortran D programs into SPMD-style Fortran code, the resulting programs were hand-instrumented to reflect the optimizations that would be performed using our combined analysis. All programs were then compiled with -O4 option of *if77* compiler (Release 4.0) for the Intel iPSC/860. We ran our tests on 16 and 32 processors of the iPSC/860 hypercube at Rice University that has 8Mb of memory per node and runs Release 3.3.2 of the Intel software. Average execution times over 10 runs for each of the programs (measured using *dclock()* microsecond timer) are reported in Table 1.

Optimizations made possible by the improved communication analysis reduced the total execution time by up to 20% compared to the programs compiled with the current Fortran D compiler prototype. In both of our benchmarks the computation is of the order $\Theta(n^2)$, and the communication is of the order $\Theta(n)$, where n is the problem size. Therefore, with the

Program	Problem Size	Proc	Execution Time in Seconds		Improvement
			FortD Compiler	Combined Analysis	
LIVERMORE 18	128×128	32	0.017750	0.014061	20.8%
		16	0.022239	0.018845	15.3%
Explicit	256×256	32	0.044297	0.036262	18.1%
Hydrodynamics		16	0.062649	0.055683	11.1%
SHALLOW	128×128	32	0.011121	0.009621	13.5%
		16	0.016393	0.014395	12.2%
Weather	256×256	32	0.029806	0.027128	9.0%
Prediction		16	0.050530	0.047693	5.6%

Table 1 Intel iPSC/860 execution times for LIVERMORE 18 and SHALLOW.

increase in problem size, the communication takes a smaller portion of the total execution time, and the impact of our communication optimizations becomes less apparent. Similarly, with the decrease in number of processors, percentage of performance improvement due to our combined analysis also decreases, because the computation time, which is not affected by our optimizations, represents more significant part of the total execution time.

There are two important ways in which the proposed analysis can improve the performance: elimination of redundant communication and hiding latency by overlapping communication with computation. Breakdown of improvements due to each of these factors is given in Table 2.

It should be noted that the Fortran D compiler already eliminates many communication redundancies that would be naively inserted by a less aggressive compiler. Therefore, although our further analysis reduces the number of communication statements in Shallow from 26 to 17 (per iteration), we only eliminate single-element messages used for periodic continuation. These messages can have significant impact only when the number of processors is large and the problem size is small, because they can cause load imbalance. With the increase in either the problem size or the number of processors, more significant portion of the overall improvement comes from the communication/computation overlap. On the other hand, eliminated messages in Livermore 18 shift whole rows of boundary elements between neighboring processors, and since the size of data communicated grows with the problem size, the performance gain due to the elimination of these messages remains more or less constant.

Without making any generalizations based on our limited experience, we would like to point out that opportunities for eliminating partly redundant communication, as described in Section 3.5, did not come up in either of the two benchmarks we analyzed; instead, all the performance gain was achieved through methods described in Sections 3.1, 3.2, and 3.4.

While analyzing possibilities for overlapping communication with computation we have run into an interesting phase-ordering problem concerning the implementation of *message aggregation*. This optimization tries to combine multiple messages corresponding to different arrays into a single message, in an attempt to reduce the startup overhead. Although aggregation might introduce extra buffering cost, it is claimed in [20] that this optimization

Program	Problem Size	Proc	Elim. Redund. Comm.	Comm/Comp. Overlap
Livermore 18	128×128	32	38.0%	62.0%
		16	33.3%	66.7%
Explicit	256×256	32	39.8%	60.2%
Hydrodynamics		16	36.9%	63.1%
Shallow	128×128	32	43.7%	56.3%
		16	26.2%	73.8%
Weather	256×256	32	22.2%	77.8%
Prediction		16	19.6%	80.4%

Table 2 Percentages of improvement due to elimination of redundant communication and overlapping communication with computation.

is always profitable if individual messages are not contiguous, because the buffering is needed anyway. However, the Fortran D compiler performs aggregation at the level of RSDs after vectorization and coalescing. Vector message pipelining, which tries to separate SENDs and RECVs, then works with these aggregated RSDs, and this significantly limits its applicability. For example, SEND for an aggregated RSD containing elements of arrays `a` and `b` would be blocked by a definition of elements of array `a`. However, without the aggregation, we would at least be able to move the SEND of `b` across this definition, and thus achieve some latency hiding.

If aggregation of RSDs might prevent opportunities for overlapping communication and computation, it seems reasonable that this optimization should be applied after our data-flow analysis. However, while this might be straightforward for messages that have both SEND and RECV at the same point in the program, in general it is a non-trivial task. Since each SEND can have multiple RECVs (in Figure 3 SEND ③ has corresponding RECVs ④ and ⑤), and vice versa, ensuring that the aggregation of SENDs goes together with the aggregation of RECVs can be quite complicated. For this reason we did not perform message aggregation on our optimized programs, but instead separate messages were generated much like in Figure 3.

5 Related Work

Several researchers have tried to optimize communication placement beyond the traditional methods based on dependence analysis. Granston and Veidenbaum apply combined flow and dependence analysis to detect redundant global memory accesses in parallel programs [8]. Using global flow analysis of array regions they are able to eliminate these redundancies, but since their global read and write are monolithic operations, they do not try to overlap communication with computation.

Communication optimization described by Amarasinghe and Lam in [2] is based on the *last write tree* representation. While they claim that their technique provides exact data-flow analysis on individual array elements, they cannot handle arbitrary control flow (loops within conditional statements, such as those in Figure 3, are not allowed), and they only optimize communication within a single loop nest.

Gong et al. describe a data-flow analysis algorithm that propagates array portions in order to determine communication placement [7]. They combine multiple communication optimizations, but their technique has several restrictions: they handle only singly nested loops and one-dimensional arrays, and their propagation algorithm is based on the structured control flow. Furthermore, although they try to overlap communication with computation, their algorithm only produces locations where SEND statements should be placed, while RECV statements "should be inserted at the points where the data are actually needed". If communication is hoisted out of the loop or if a message is split, as suggested, into several SEND statements, simply inserting a RECV just before the data are needed is not satisfactory. In contrast, our approach provides placement points for both SENDs and RECVs, which are provably balanced [13].

In the most recent work on optimizing communication, Gupta et al. show how partial redundancy elimination can be applied to *available section descriptors* [11]. The available section descriptor extends an array section descriptor with the mapping of the array section

onto the virtual processor grid [10]. Since we opt for the efficiency of bit-vector based flow analysis, and analyze array sections only in the initialization phase, it is possible that their method will be more precise. However, they do not present any experimental data that would indicate whether the cost of their analysis would be justified in practice by the need for extra precision. Much like in [7], their SENDs are performed "as early as legally possible", but RECVs (or rather WAITs for non-blocking RECVs) are inserted "at the reference to non-local data", causing the same problem as discussed above.

6 Conclusions

We have presented a method for optimizing communication when compiling HPF-like languages. This method, based on the combination of dependence and data-flow analyses, allows us to perform message vectorization, elimination of redundant messages, and overlapping of communication with computation. For the sake of efficiency, we use bit-vector based analysis and satisfactory precision is achieved by examining relationships among array portions when initializing the data-flow framework. Our preliminary experience, though limited in scope, indicates that optimizations based on the proposed method can result in significant performance improvement.

As mentioned in Section 4 we have modified the existing implementation of the GIVE-N-TAKE framework so that array portions representable with RSDs are analyzed when initializing data-flow variables. Dependence analysis will be fully integrated with the code placement framework once the design and implementation of the new set representation are finished. This new representation will have to support communication sets that can be created with `BLOCK_CYCLIC` and multidimensional distributions. We also plan to investigate whether data-flow analysis that would use this set representation for all lattice operations could be more profitable and how big an increase in compile time it would cause.

Communication optimizations described in this paper do not involve program transformations. However, since aggressive loop transformations (interchange, strip-mining, distribution, fusion) can often improve program's performance, the interaction of these transformations and our analysis techniques needs to be studied further.

Finally, placing SENDs as early as possible, and RECVs as late as possible, can potentially have a negative effect of keeping message buffers full for unnecessarily long time. This problem could be handled in a postprocessing pass, that would take performance estimation and machine parameters into account in order to move SENDs closer to their corresponding RECVs, while still overlapping communication with computation.

Acknowledgments

The authors would like to thank Debbie Campbell and Ajay Sethi for their helpful comments on an earlier draft of this paper. The iPSC/860 that was used for the experimental studies in the paper was purchased with funds from NSF Institutional Infrastructure Grant CDA-86198393 and support from the Keck Foundation.

References

[1] V. Adve, C. Koelbel, and J. Mellor-Crummey. Performance analysis of data-parallel programs. Technical Report CRPC-TR94405, Center for Research on Parallel Computation, Rice University, May 1994.

[2] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[3] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[4] H. Berryman and J. Saltz. A manual for PARTI runtime primitives. ICASE Interim Report 13, Institute for Computer Application in Science and Engineering, Hampton, VA, September 1990.

[5] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran — A Fortran language extension for distributed memory multiprocessors. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[6] M. Gerndt. Updating distributed variables in local computations. *Concurrency: Practice and Experience*, 2(3):171–193, September 1990.

[7] C. Gong, R. Gupta, and R. Melhem. Compilation techniques for optimizing communication on distributed-memory systems. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[8] E. Granston and A. Veidenbaum. Detecting redundant accesses to array data. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[9] T. Gross and P. Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software—Practice and Experience*, 20(2):133–155, February 1990.

[10] M. Gupta and E. Schonberg. A framework for exploiting data availability to optimize communication. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[11] M. Gupta, E. Schonberg, and H. Srinivasan. A unified data-flow framework for optimizing communication. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[12] R. v. Hanxleden. Handling irregular problems with Fortran D — A preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[13] R. v. Hanxleden and K. Kennedy. A code placement framework and its application to communication generation. Technical Report CRPC-TR93337-S, Center for Research on Parallel Computation, Rice University, October 1993.

[14] R. v. Hanxleden and K. Kennedy. Give-N-Take — A balanced code placement framework. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[15] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[16] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[17] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[18] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, July 1992.

[19] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[20] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, January 1993.