

Combining Dependence and Data-Flow Analyses to Optimize Communication

Ken Kennedy
ken@rice.edu

Department of Computer Science, Rice University

Nenad Nedeljković
nenad@rice.edu

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

Data-parallel languages, such as Fortran D [9] and High Performance Fortran (HPF) [10], are designed to facilitate writing of portable programs for multi-computers. They provide shared address space and means for the programmer to specify how data should be distributed among processors. Using the distribution directives, compilers for these languages generate SPMD programs in which the computation is partitioned across processors and the communication inserted where necessary. The main obstacle in achieving high performance when running these programs on distributed-memory machines is the high cost of inter-processor communication relative to the cost of accessing local data. Therefore, it is very important to reduce the number of messages inserted by the compiler.

In the Fortran D compiler prototype developed at Rice University, communication resulting from regular array references (those where subscripts are linear combinations of loop index variables) is optimized based primarily on dependence analysis [11]. The effectiveness of optimizations, the most important of which are message vectorization and coalescing, is limited by the fact that most of the analysis is performed for a single loop nest at a time, and little is done to optimize communication across arbitrary control flow.

On the other hand, communication due to irregular array accesses (those where subscripts are not linear combinations of loop index variables) is optimized via the GIVE-N-TAKE data-flow framework [7]. Although this framework provides global analysis on the control flow graph, it treats arrays as indivisible units and does not exploit 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 sections accessed, and data-flow analysis based on the GIVE-N-TAKE framework, to propagate this information across arbitrary control flow. The combined approach enables 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 optimizing communication [4, 3, 5]. Most have focused on extending the existing methods to work with array section descriptors. In contrast, our data-flow analysis uses bit vectors (each bit represents an array portion, which is a set of elements that either need to be communicated, or whose definition affects the communication) and is likely to be more efficient. We maintain high level of precision by examining the relations 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 paper is organized as follows: In Section 2 we describe two current methods for optimizing communication. Our technique for combining dependence and data-flow analyses is presented in Section 3 and initial experience in Section 4. We discuss related work in Section 5 and conclude in Section 6 by summarizing our contributions and outlining directions for future work.

2 Motivation

We now describe some drawbacks of the dependence-based approach, which is used to optimize regular communication in the old Fortran D compiler. After that, we give an outline of the data-flow framework used for irregular array accesses and present problems that arise when the same analysis is applied to regular references.

2.1 Optimizing regular communication

The most important optimization for regular array 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 multiple single-element messages are replaced with a single vectorized message [11].

In order to avoid redundant communication, the compiler applies *message coalescing*, combining 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 many opportunities for eliminating redundant messages are missed.

Overlapping communication and computation is an important technique for improving 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 Sends and Recvs towards their definitions and uses respectively. However, the support for this optimization in the existing compiler prototype is very limited.

The program in Figure 1 illustrates the communication placement for regular references. The original HPF-like program is shown on the left. Since there are 4 processors, each one gets 10 elements of each of the arrays a, b, and c. Based on this distribution, the compiler partitions the computation (using the *owner computes* rule) and inserts communication, as shown on the right. For example, statement $\text{Comm } 1 \xrightarrow{b(11)}_0$ means that the array element b(11) is sent from processor 1 to processor 0. Note that loops and communication in the SPMD node code are also given in global indices.

In our example, communication ③ is clearly redundant, since the same elements of array b have already been communicated in ①, in order to satisfy the non-local references to b in the first do loop. Because message coalescing is performed only within a single loop nest, this redundant communication is not removed.

Communication ④ 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. It is, therefore, 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.

2.2 Give-N-Take framework

Typical examples of irregular array accesses are those where a subscript itself is a reference to an indirection array. In these cases it is hardly possible to extract significant compile-time knowledge about elements accessed, and dependence analysis is of little use. Instead, the Fortran D compiler's analysis is based on the GIVE-N-TAKE code placement framework [7]. This framework uses a producer-consumer concept where

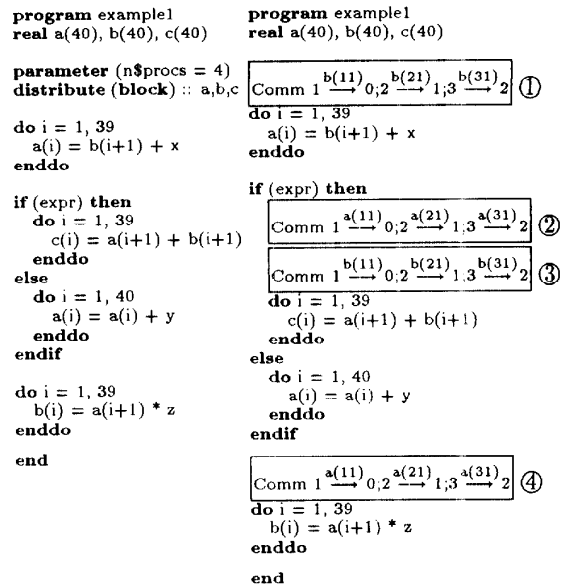


Figure 1 Regular communication placement.

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. In addition, the framework takes into account that data may also be destroyed, i.e., redefined, or given for free through side effects.

For each node in the program's 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. Data-flow equations are then evaluated to propagate this information globally. The result of the analysis is given by output variables that indicate where the production of data should be placed. If only references to non-local data need to be satisfied, 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, then we also need to find placement for global Writes (Scatters), 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) so that its start (Send) is placed as early as possible and its end (Recv) as late as possible.

While the GIVE-N-TAKE framework is in principle applicable to regular references, its current implementation has several limitations, 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 pro-

<pre> program example2 real a(40), b(40), c(40) parameter (n\$procs = 4) distribute (block) :: a,b,c do i = 2, 40, 2 a(i) = x enddo do i = 1, 39 b(i) = a(i+1) * y enddo do i = 1, 39 do j = 1, 10 c(i) = c(i) + c(i+1) * b(j) enddo enddo end </pre>	<pre> program example2 real a(40), b(40), c(40) do i = 2, 40, 2 a(i) = x enddo Read Send/Recv a(2:40) ① do i = 1, 39 b(i) = a(i+1) * y enddo Read Send/Recv b(1:10) ② do i = 1, 39 do j = 1, 10 Read Send/Recv c(i+1) ③ c(i) = c(i) + c(i+1) * b(j) enddo enddo end </pre>
--	--

Figure 2 GIVE-N-TAKE communication placement.

gram [6]. For irregular problems, determining which of these references require communication is done at run time through calls to PARTI/CHAOS library routines (Gather) [2]. However, in regular codes it is often possible to extract some static information and determine which references are non-local at compile time.

When comparing two portions of the same array, the compiler must assume the most conservative facts if these portions are accessed via irregular subscripts. If we need to decide whether communication of an array portion can be hoisted across a definition of the other portion, we must find out if the two portions intersect. If these are irregular portions of the same array, at compile time we must assume that they can interfere with each other, since otherwise we might place communication so 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 with regular array references. Communication ①, which performs a global Read for a(2:40), 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, because they both use the same array a. Moreover, even if the compiler compared regular array sections a(2:40:2) (defined in the first loop) and a(2:40) (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 referenced array portion 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), 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 communication ③ is not. The reason for this is that array references are not analyzed on a per element basis, but instead it is assumed that the same portion of array c is both defined and used within the loop nest. However, the method based on the level of loop-carried true dependencies would place this communication at the same location as ②.

3 Combined approach

We now show how to combine the two analyses described in the previous section 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 dependencies.

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 program.

The Fortran D compiler prototype uses *regular section descriptors* (RSDs) to represent the sets of array elements that need to be communicated [8]. These RSDs are augmented to handle simple forms of boundary conditions [11]. 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 two 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.

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 definitions of non-owned data are not allowed, 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:

$$\begin{aligned} \text{Read.TAKE}_{init} &:= \text{Comm}, \\ \text{Read.STEAL}_{init} &:= \text{affects}(\text{Def}), \\ \text{Read.GIVE}_{init} &:= \text{contains}(\text{Comm}). \end{aligned}$$

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 [6], 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 yield 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 $a(1:40)$ 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.

When checking if one communication subsumes another, it is not enough to look only at the data communicated, but we also need to take into account processors participating in the communication. Given a shift communication $\text{Comm } 1 \xrightarrow{a(11)}_0; 2 \xrightarrow{a(21)}_1; 3 \xrightarrow{a(31)}_2$ and a broadcast $\text{Comm } 1 \xrightarrow{a(11)}_{0,2,3}$, 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 ($a(11, 21, 32)$). 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 4 processors.

Once the input variables are initialized, we proceed with the GIVE-N-TAKE analysis. All lattice operations needed to evaluate data-flow equations are performed on bit vectors. It would be possible to modify the framework so that lattice operations are done on array portions themselves, instead of using just bits that represent those 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 since we only look at the relations among array sections when initializing the framework and during the data-flow analysis perform simple logical operations on bit vectors. On the other hand, evaluating intersection, union and difference of array portions can be 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.

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.

Distribution directives from the original program are not shown in Figure 3, but instead it is assumed that all arrays are perfectly aligned and **block**-distributed over 4 processors. As in previous examples, loop bounds are given in their original form, even though they will be reduced when the compiler partitions the computation.

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 TAKE sets are initialized to contain only array portions that require communication, annotations to the control-flow graph are propagated to achieve the placement of Sends and Recvs shown on the right. 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.

Separation of Sends and Recvs has also been done for communication ⑥ (Send - ⑥, Recv - ⑧), and communication ⑦ (Send - ①, Recv - ⑨). Note 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(21)) does not intersect the array section communicated (a(11)). However, statement ③ could not be moved any further up, since the first do loop defines the array elements that are communicated.

3.4 Discussion

While the optimizations described in Section 3.3 represent an improvement to the existing Fortran D compiler, there is still some communication redundancy left. A part of communication ⑦ on the right side of Figure 3 is redundant, because the communication $1 \xrightarrow{a(11)}_0$ had already taken place (Send - ③, Recv - ④, ⑤), and array element a(11) was not re-defined. Therefore, only the communication $2 \xrightarrow{a(21)}_1$ 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 data-flow analysis. We do not analyze relations 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_{inst}).

It is not clear if partly redundant communication occurs frequently in real programs. Moreover, elimination of these redundancies does not necessarily reduce the execution time. In our example, the cost of communicating $1 \xrightarrow{a(11)}_0$; $2 \xrightarrow{a(21)}_1$ should be practically the same

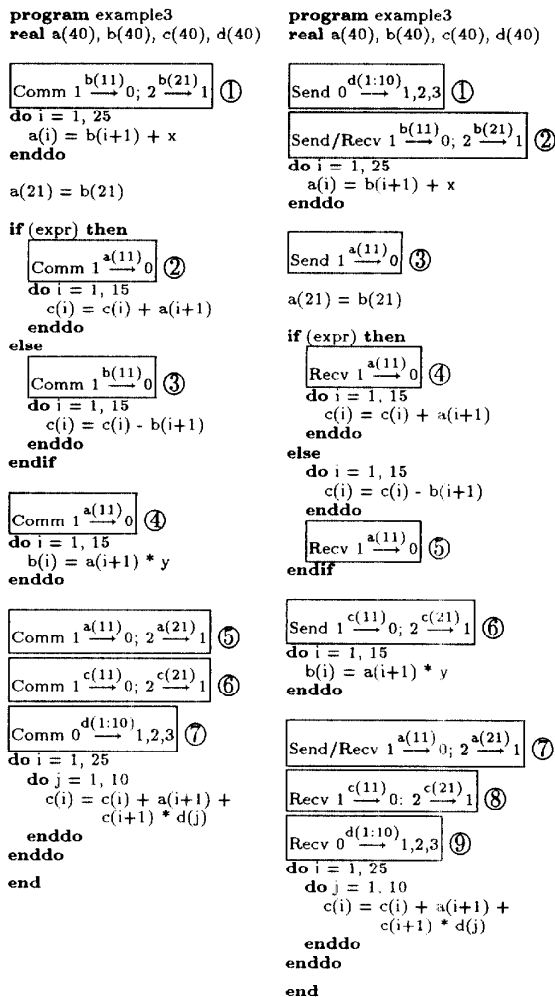


Figure 3 Left: Initial communication annotations. Right: Message placement based on data-flow analysis.

as the cost of $2 \xrightarrow{a(21)}_1$, since 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 [3], but we only do it when initializing the data-flow framework.) Since we are only interested in control-flow reachability without taking array kill information into account, this ap-

proach 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, 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 $1 \xrightarrow{c(11)}_0$ and $2 \xrightarrow{c(21)}_1$), and the part $Send\ 2 \xrightarrow{c(21)}_1$, would be moved all the way to the beginning of the program.

4 Preliminary experience

We have modified the existing implementation of the GIVE-N-TAKE framework [6] to include the support for array portions that can be represented with RSDs. Although the framework can now take advantage of compile-time knowledge about array elements accessed, integration with dependence analysis is not yet complete. However, we were able to run an experiment to measure the potential benefits of our approach.

We used LIVERMORE 18 explicit hydrodynamics kernel and SHALLOW weather prediction program written by Paul Swartztrauber (NCAR). Both benchmarks are highly data-parallel and significant speedups were reported with the existing Fortran D compiler [11]. 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.

After translating original Fortran D programs into SPMD node 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 and run on 16 and 32 processors of an Intel iPSC/860 hypercube. Execution times in milliseconds (measured using *dclock* 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 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.

The proposed technique improves the performance in two ways: redundant communication is eliminated, and latency is hidden by overlapping communication with computation. Breakdown of percentages of total improvement due to these factors is given in Table 2.

The Fortran D compiler eliminates many redundant messages that would be naively inserted by a less aggressive compiler. Thus, although our further analysis reduces the number of messages in SHALLOW from 26 to 17 (per iteration), we only eliminate single-element messages used for periodic continuation. Their impact is significant only when the number of processors is large and the problem size is small, because they cause load imbalance. With the increase in either the problem size or the number of processors, more significant portion of the total improvement comes from the communication/computation overlap. In contrast, 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 3.4, 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 and 3.2.

Program	Size	# Proc.	milliseconds		% Diff.
			Fortran D	Improved	
Liv. 18	128	32	17.75	14.06	20.8
		16	22.24	18.84	15.3
	256	32	44.30	36.26	18.1
		16	62.65	55.68	11.1
Shallow	128	32	11.12	9.62	13.5
		16	16.39	14.40	12.1
	256	32	29.81	27.13	9.0
		16	50.53	47.69	5.6

Table 1 Timings for LIVERMORE 18 and SHALLOW.

Program	Size	Proc.	Redundancy	Overlap
Liv. 18	128	32	38.0%	62.0%
		16	33.3%	66.7%
	256	32	39.8%	60.2%
		16	36.9%	63.1%
Shallow	128	32	43.7%	56.3%
		16	26.2%	73.8%
	256	32	22.2%	77.8%
		16	19.6%	80.4%

Table 2 Breakdown of percentages of total improvement due to eliminating redundant communication and overlapping communication with computation.

5 Related work

Several researchers have tried to optimize communication beyond the traditional methods based on dependence analysis. Granston and Veidenbaum use flow analysis of array regions to detect and eliminate redundant global memory accesses [4]. 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 is based on the *last write tree* representation [1]. They do not handle arbitrary control flow, e.g., loops inside conditional statements, and optimize communication only within a single loop nest.

Gong et al. describe a data-flow analysis technique that unifies multiple communication optimizations [3], but they only handle singly nested loops and one-dimensional arrays. Although they try to overlap communication with computation, their algorithm only produces the placement of Send statements; if care is not taken in placing Recv statements this can lead to unbalanced communication.

Gupta et al. show how partial redundancy elimination can be applied to *available section descriptors* [5]. Since we opt for the efficiency of bit-vector flow analysis, it is possible that their method will be more precise. However, they do not present any experimental data to show if the cost of their analysis would be justified in practice by the need for extra precision. Much like in [3], they only find the placement of Sends, and not Recvs, facing 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 and computation. We use efficient bit-vector analysis, but achieve high precision by examining relations among array portions when initializing the data-flow framework. Our preliminary experience, though limited in scope, indicates that optimizations based on the proposed technique 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 (that will support communication sets that can be created with **block-cyclic** and multidimensional distributions) are completed.

Communication optimizations described in this paper do not involve loop transformations, such as inter-

change, strip-mining, distribution, and fusion. Since they can often improve the program's performance, interaction of these transformations and our analysis techniques needs to be studied further.

Finally, the policy of initiating communication as early as possible ignores constraints imposed by hardware resources, e.g., message buffers. Further refinement of the framework should take performance estimation and machine parameters into account when making decisions about the communication placement.

Acknowledgments

We thank Debbie Campbell and Ajay Sethi for their help in proofreading the paper. This work was supported in part by ARPA contract DABT63-92-C-0038 and NSF Cooperative Agreement #CCR-9120008.

References

- [1] 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.
- [2] 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.
- [3] 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.
- [4] E. Granston and A. Veidenbaum. Detecting redundant accesses to array data. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.
- [11] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, January 1993.